7-5-1990

Modular Verification of Object-Oriented Programs with Subtypes

Gary T. Leavens
Iowa State University

Modular Verification of Object-Oriented Programs with Subtypes

Abstract
Object-oriented programming languages like Smalltalk-80 have a message passing mechanism that allows code to work on instances of many different types. Techniques for the formal specification of such polymorphic functions and abstract types are described, as well as a logic for verifying programs that use message passing but not object mutation or assignment. The reasoning techniques formalize informal methods based on the use of subtypes. A formal definition of subtype relationships among abstract types whose objects have no time-varying state but may be nondeterministic or incompletely specified is given. This definition captures the intuition that each instance of a subtype behaves like some instance of that type's supertypes. Specifications of polymorphic functions are written by allowing instances of subtypes as arguments. Restrictions on the way that abstract types are specified ensure that such function specifications are meaningful and do not have to be rewritten when new subtypes are specified. Verification consists of showing that the specified relation among types has certain semantic properties, that each expression's value is an instance of a subtype of the expression's type, and a proof of correctness that ignores subtyping.

Keywords
programming languages, object-oriented, Smalltalk, specification, verification, subtype, type checking, abstract type, message passing, polymorphism

Disciplines
Systems Architecture

Modular Verification of Object-Oriented Programs with Subtypes

Gary T. Leavens

TR #90-09
July 5, 1990

Department of Computer Science
Iowa State University
Ames, Iowa 50011-1040, USA

Modular Verification of Object-Oriented Programs with Subtypes

Gary T. Leavens

Abstract

Object-oriented programming languages like Smalltalk-80 have a message passing mechanism that allows code to work on instances of many different types. Techniques for the formal specification of such polymorphic functions and abstract types are described, as well as a logic for verifying programs that use message passing but not object mutation or assignment.

The reasoning techniques formalize informal methods based on the use of subtypes. A formal definition of subtype relationships among abstract types whose objects have no time-varying state but may be nondeterministic or incompletely specified is given. This definition captures the intuition that each instance of a subtype behaves like some instance of that type's supertypes. Specifications of polymorphic functions are written by allowing instances of subtypes as arguments. Restrictions on the way that abstract types are specified ensure that such function specifications are meaningful and do not have to be rewritten when new subtypes are specified. Verification consists of showing that the specified relation among types has certain semantic properties, that each expression's value is an instance of a subtype of the expression's type, and a proof of correctness that ignores subtyping.

Keywords: programming languages, object-oriented, Smalltalk, specification, verification, subtype, type checking, abstract type, message passing, polymorphism.

1987 CR Categories:

D.2.1 [*Software Engineering*] Requirements/Specifications — Languages; D.2.4 [*Software Engineering*] Program Verification — Correctness proofs; D.3.3 [*Programming Languages*] Language Constructs — Abstract data types, procedures, functions, and subroutines; F.3.1 [*Logics and Meanings of Programs*] Specifying and verifying and reasoning about programs — logics of programs, pre- and post-conditions, specification techniques.

List of Figures

Contents

Preface

This report is a complete revision of my doctoral dissertation [Lea88] [Lea89]. As such it differs in most of the technical details, although it has the same goals and is based on roughly the same plan for specification and verification of object-oriented programs.

My deep thanks go to Bill Weihl, who not only helped me develop the ideas, but who is also an excellent reader and critic, a good friend, and has also aided in this revision. Barbara Liskov was also instrumental in developing and organizing my ideas and she encouraged me to pursue the topic of subtyping. During the writing of my dissertation, John Guttag provided technical expertise, encouragement, and a dedication to clear writing and thinking, which I hope is reflected in the present work.

For specific technical suggestions reflected in this revision, thanks are due to the following. John Guttag, at my defense, suggested that I require that trait functions as well as the generic operations apply at each subtype; Bill Weihl provided some examples that helped convince me of the wisdom of that suggestion. Jeannette Wing pointed out related work by Reynolds that helped formalize Guttag's suggestion. David McAllester crystallized an idea of mine in his suggestion that I base the definition of subtype relations on an algebraic criterion instead of observations (as was done in my dissertation). I had stimulating technical discussions with many others, including Val Breazu-Tannen, John Mitchell and William Cook.

For great love, understanding, and encouragement, my thanks to Janet Leavens. For help, friendship and encouragement during the past few years, my thanks to Brian Oki, Kelvin Nilsen, Albert Baker, Arch Oldehoeft, the faculty and staff at Iowa State, Jim Bieman, and my family.

This work was supported in part by the ISU Achievement Foundation, the National Science Foundation under Grants DCR-8510014 and CCR-8716884, the Defense Advanced Research Projects Agency under Contract N00014-83-K-0125, and by a GenRad/AEA Faculty Development Fellowship.

Chapter 1

Introduction

The ability to easily expand the functionality of a software system is fundamental to the maintenance and prototyping of computer software. An important way that one can enhance the functionality of a system is by adding new types of objects. The message passing mechanism of an object-oriented programming language such as Smalltalk-80 [GR83], C++ [Str86], the Common LISP Object System (CLOS) [Kee89], or Simula 67 [BDMN73], can eliminate the need to update code to work with objects of new types [Cox86], because it separates the manipulation of an object from knowledge of the object's type.

To reason about the behavior of a program to which new types of objects have been added, programmers often classify types by how instances of that type behave. If each instance of type `S` behaves like an instance of type `T`, then `S` is called a subtype of `T`. Programmers hope that if their code works correctly when it operates on instances of some type `T`, then their code will also work correctly when a new subtype of `T` is added to the program. However, since they lack the formal tools to guide their classification of types and their reasoning, their reasoning lacks certainty.

The following steps towards a foundation for reasoning about programs that use message passing and subtyping are described in this report:

- a formal technique for specifying programs that use message passing,
- a formal definition of subtyping among abstract types, and
- a formal technique for verifying programs that use message passing.

Even if formal verification of such programs is not practical, large programs require at least careful informal specifications of their modules. A better understanding of formal techniques for specification and verification can serve as a guide to such informal reasoning. Furthermore, correctness considerations play an important role in program optimization.

The remainder of this chapter contains a discussion of the purpose and background of the research, an overview of the solution, a survey of related work, and a guide to the technical details.

1.1 Purpose and Background

Object-oriented design techniques arise out of concerns for modularity in software systems [Cox86]. Modularity means the separation of code into modules that have few and well-defined interactions with each other. Modules can be used to hide design decisions that may have to be changed at a later time [Par72], allowing one to view a system's design at many different levels of detail. This aids the construction of software systems as well as their maintenance, since when a design decision is changed, only one module should need to be changed.

An object-oriented design (as opposed to a program) consists of procedure and abstract data type specifications that, when implemented and put together, will satisfy the system's requirements [LG86, Page 265]. The implementation of a procedure or abstract data type specification may itself be guided by a lower-level design. An *abstract data type* is specified by describing a set of *objects* (the instances of that type) and how those objects behave when manipulated by the type's *operations*. For example, `Int` is an abstract data type, whose objects are integers and whose operations are `add`, `mul`, and so on.

The behavior of the objects of an abstract type is described in a *type specification*, which does not dictate the details of an implementation. For example, the specification of the type `IntSet` describes the behavior of the following operations (see Figure 1.1):

- `null`, which creates an empty `IntSet`,
- `ins`, which returns an `IntSet` containing its integer argument inserted into the set of elements of its `IntSet` argument,
- `elem`, which tests whether an integer is in an `IntSet`,
- `choose`, which returns an arbitrary element of a nonempty `IntSet`,
- `size`, which returns the size of an `IntSet`, and
- `remove`, which returns an `IntSet` containing all the elements of its `IntSet` argument except for its integer argument.

None of the operations of `IntSet` changes the state of an existing `IntSet`. Since the instances of type `IntSet` have no time-varying state they are *immutable*. A type whose objects are all immutable is itself said to be `immutable`, as will be all the types discussed below.

Although designs do not change or wear out by themselves, they must still be maintained, because the world around them changes. If the design has made careful use of abstract data types, then it will be relatively easy to change design decisions, especially decisions about the data structures used to represent objects [Par72]. However, even a careful design using

IntSet **immutable type**
class ops [null]
instance ops [ins, elem, choose, size, remove]
based on sort C **from** IntSetTrait

op null(c:IntSetClass) **returns**(s:IntSet)
ensures s == {}

op ins(s:IntSet, i:Int) **returns**(r:IntSet)
ensures r == s ∪ {i}
op elem(s:IntSet, i:Int) **returns**(b:Bool)
ensures b = (i ∈ s)
op choose(s:IntSet) **returns**(i:Int)
requires ¬ isEmpty(s)
ensures i ∈ s
op size(s:IntSet) **returns**(i:Int)
ensures i = toInt(size(s))
op remove(s:IntSet, i:Int) **returns**(r:IntSet)
ensures r == delete(s,i)

Figure 1.1: The type specification `IntSet`.

abstract data types alone may not be easily allow the functionality of a software system to be enhanced by adding new types of objects to the system.

As an example of enhancement of a system's functionality, consider a system to keep track of which keys unlock certain doors and the keys possessed by various people. One can represent key numbers by objects of type `Int` (integers), and the set of keys possessed by a person as an object of type `IntSet`. (There would be other types as well.) The operations of these types can be used to perform such tasks as recording issued and returned keys, and finding whether two people have any keys in common.

Later, one might want to extend the keys system to issue a set of keys with consecutive numbers. Since such a set can be represented in less storage than a general set, it may be wise to add a new type to the design. This is the type `Interval` (see Figure 1.2). The operations of the type `Interval` are the same as those for `IntSet`, except that instead of `null` there is an operation `create` that takes two integer arguments and returns an `Interval` object representing all the integers between the arguments (inclusive). The integer arguments must be ordered. The `ins` and `remove` operations of `Interval`s may return either objects of type `IntSet` or `Interval`, depending on their arguments. The `choose` operation of `Interval`s will always return the least element of the `Interval`. The type `Interval` is intended to be a subtype of `IntSet`, since objects of type `Interval` can be treated as `IntSet` objects without showing surprising behavior.

When one adds an implementation of the type `Interval` to the program, one must update the code to work with `Interval` objects and ensure the correctness of the updated code.

Interval **immutable type**
subtype of IntSet
by $[l, u]$ **simulates** toSet($[l, u]$)
class ops [create]
instance ops [ins, elem, choose, size, remove]
based on sort C **from** IntervalTrait

op create(c:IntervalClass, lb,ub:Int)
returns(i:Interval)
requires lb ≤ ub
ensures i == [lb,ub]

op ins(s:Interval, i:Int) **returns**(r:IntSet)
ensures r == s ∪ {i}
op elem(s:Interval, i:Int) **returns**(b:Bool)
ensures b = (i ∈ s)
op choose(s:Interval) **returns**(i:Int)
ensures i = leastElement(s)
op size(s:Interval) **returns**(i:Int)
ensures i = toInt(size(s))
op remove(s:Interval, i:Int) **returns**(r:IntSet)
ensures r == delete(s,i)

Figure 1.2: The type specification `Interval`.

1.1.1 The Code Update Problem

The code update problem is demonstrated by the function `inBothCLU` of Figure 1.3, which is written in CLU[LAB+81]. If the `IntSet` arguments of `inBothCLU` have an element in common, then the function returns an integer in their intersection (see Figure 1.6 for a specification). The notation `IntSet$choose` in a CLU program means the `choose` operation of the module that implements `IntSet`. Although the code for `inBothCLU` is correct for arguments of type `IntSet`, it must be changed to work with objects of type `Interval`.

Message Passing and Method Dictionaries. The message passing (or late binding) mechanism of an object-oriented programming language can eliminate the need to update code to work with objects of new types [Cox86]. Each object in such a language contains[1] both data and a table of operations called a *method dictionary*. An object's method dictionary maps message names to the operations (code) that implements the named operations for a given type [WB89]. Since the method dictionary is accessible from the objects, code that invokes an object's operations does not have to depend on the types of objects. For example, one does not write `IntSet$ins(s,e)` to insert an integer `e` into a set `s`, as one would in CLU;

[1] For space efficiency, in most implementations of object-oriented languages, the code for an operation is shared among all objects of the same type.

```
inBothCLU = proc(s1,s2: IntSet)
        returns(i:Int)
    return(testForCLU(IntSet$choose(s1),
           s1, s2))
end inBothCLU

testForCLU = proc(i,s1,s2: IntSet)
        returns(i:Int)
    if IntSet$elem(s2,i)
    then return(i)
    else return (testForCLU(
        IntSet$choose(
            IntSet$remove(s1,i)),
            IntSet$remove(s1,i), s2))
    end
end testForCLU
```

Figure 1.3: Implementation of the function `inBothCLU` in CLU, which demonstrates the code update problem.

instead, one writes `s.ins(e)` (in Simula 67 or C++) to insert `e` into `s`, which invokes the operation `ins` from the object `s`. Thus *message passing* means fetching an object's operation from its method dictionary and invoking it. (Message passing is sometimes called late binding, dynamic binding, or generic invocation. Message passing is not necessarily concurrent.) Metaphorically `s.ins(e)` means "send the message named `ins` with argument `e` to the object denoted by `s`." The advantage of using message passing is that the call `s.ins(e)` can invoke the `ins` operation of the types `IntSet`, `Interval`, or even types that have not yet been imagined. Thus if the original keys system was written using message passing, then adding the new type `Interval` does not necessitate rewriting of existing code.

In what follows, a slightly more general notation will be used for message passing. That is, `ins(s,e)` will mean sending the message `ins` to `s` and `e`. The exact procedure that is invoked will depend on the types of both objects, `s` and `e`, as in CLOS [Kee89]. Often it will be appropriate to think of `ins(s,e)` as sending the message `ins` with argument `e` to the object denoted by `s`. But the notation `ins(s,e)` better reflects the possibility that the procedure invoked may depend on the types of all arguments, instead of just the type of `s`. This is important in dealing with binary operations such as `union` or `intersection` (although these are not part of the type `IntSet` as specified above). To avoid confusion with function calls (which do not use message passing and are thus statically bound), function names will be written in a slanted font: *funName*, while message names are written in a typewriter font: `msgName`.

Subtype Polymorphism and Dynamic Overloading. Code written using message passing is *polymorphic*, because it can produce roughly the same effect on arguments of different types. For example, a function *inBoth* can find a key number that is common to two `IntSet` objects, or two `Interval` objects (or an `IntSet` and an `Interval`) using the same sequence of message sends. This kind of polymorphism is called *subtype polymorphism* (or inclusion polymorphism [CW85]). To understand why subtype polymorphism makes reasoning about programs difficult, it is necessary to understand how subtype polymorphism differs from other kinds of polymorphism.

A polymorphic procedure must generally be supplied with a method dictionary for each type of parameter. For example, a polymorphic `sort` procedure would need a method dictionary for the type of elements it is to sort, which would provide the "$\geq$" operation to compare elements. In some languages the method dictionary is implicit in a type parameter, such as `Int` in the instantiation `sort[Int]` as one might write in CLU [LAB+81] or Ada [Ada83]. In ML [GMW79], type parameters are implicit, but one must pass the operations that would go in the method dictionary explicitly.

The polymorphism that results from forms of overloading can also be explained by method dictionaries [WB89]. An operation name, such as `+`, is *overloaded* if the same name will invoke different operations depending on the types of its arguments. For example, in a language where arithmetic operators like `+` can mean either integer or floating-point operations, one can write expressions such as `a+b`, and the compiler determines which method dictionary (integer or float) to consult for the meaning of `+`. The functional language Haskell [Hud89] allows a function with body `a+b` to be applied to arguments of different types; such a procedure is passed a method dictionary that gives meaning to the overloaded operations. In Haskell and other languages with static overloading (such as Ada) the method dictionary that must be consulted is known statically. Message passing, however, is a form of dynamic overloading, since the same message name may be used to invoke different operations; that is, with message passing the overloading is only resolved at run-time.

Subtype polymorphism is distinguished by two features from other kinds of polymorphism: the implicit association of a method dictionary with each object, and the possibility that a given expression may denote objects with different method dictionaries during different executions of the program. It is impossible, in a language with subtype polymorphism, to statically determine the appropriate method dictionary for an expression. For example, if `s1` is a formal argument that can be either an `IntSet` or an `Interval`, then until the actual argument's value is known, the method dictionary for `s1` is unknown, and hence the exact operational effect of `choose(s1)` is also unknown. This is illustrated by the program of Figure 1.4, in which there are two function definitions and a main program. The main program consists of an **if** expression that calls the function *inBoth* with different arguments. The expression `choose(s1)` in the second line (i.e., in the body of *inBoth*) is an example of a message send. This message send will invoke a procedure defined in the class `IntSet` or `Interval`, depending on the program's input. However, until the program's input is known, it cannot be said which procedures will be invoked by these message sends.

By contrast, consider writing *inBoth* in a language

```
fun inBoth (s1,s2:IntSet) =
    testFor(choose(s1), s1, s2);
fun testFor (i:Int, s1,s2:IntSet) =
    if elem(s2, i)
    then i
    else testFor(choose(remove(s1,i)),
                 remove(s1,i), s2)
    fi;
program (b:Bool):  Int =
    if b
    then inBoth(ins(null(IntSet),3),
                create(Interval,2,5))
    else inBoth(create(Interval,1,4),
                create(Interval,2,5))
    fi
```

Figure 1.4: Example of message passing.

```
letrec testForML(i,s1,s2,
            choose,remove,elem) =
    if elem(s2, i)
    then i
    else testForML(choose(remove(s1,i)),
        remove(s1,i), s2,
        choose, remove, elem) ;;

let inBothML(s1,s2,choose,remove,elem) =
    testForML(choose(s1), s1, s2,
        choose, remove, elem) ;;
```

Figure 1.5: Implementation of the function `inBothML` in ML.

with *static polymorphism*, such as ML [GMW79]. In ML, the operations `choose`, `remove`, and `elem` must be passed as arguments, because an object's operations are not implicitly available at run-time in ML [2]. (See Figure 1.5.)

The advantages of using subtype polymorphism to solve the code update problem are as follows.

- The types of arguments to a polymorphic function can be limited to subtypes of specific types. Cardelli and Wegner call this idea "bounded quantification" [CW85]. For example, the declared type of the arguments of the function *inBoth* in Figure 1.4 is `IntSet`, which means that objects of all types that are *subtypes* of `IntSet` (such as `Interval`) can be used as arguments, but objects that are not instances of a subtype of `IntSet` cannot be passed as actual arguments. This property will be useful in solving some of the specification and verification problems discussed below.

  Furthermore, a programming language's type system can statically enforce such limits on the use of polymorphic functions. For example, in Trellis/Owl [SCB+86] and C++ type checking is based on a declared subtype relationship. By contrast, if a language does not have a notion of subtype, then the language's type system cannot help enforce semantic limits on the use of polymorphic functions (unless the type system is rich enough to state such constraints directly).

- Subtype polymorphism allows more flexible behavior at run-time than other kinds of parametric polymorphism, while permitting static type checking. Without a notion of subtype relationships, a static type system must have a static knowledge of method dictionaries. For example, in ML or Ada the types of arguments must be statically known. With subtype polymorphism, exact knowledge of method dictionaries can be postponed until run-time, but one may still do static type checking [SCB+86].

- Programs may be more terse with subtype polymorphism, since instance operations are implicitly associated with objects, and thus extra arguments can be suppressed. The difference is obvious when compared to a language such as ML, where all the operations of the method dictionary have to be passed. However, in a language such as Ada or CLU, method dictionaries are associated with type parameters, so the difference between `inBoth[IntSet,Interval](s,i)`, and *inBoth*`(s,i)` is small.

1.1.2 The Specification and Verification Update Problems

How should one reason about the behavior of a program to which new types of objects have been added? For example, suppose that, before adding the type `Interval` to the keys system, one has verified that the implementation of *inBoth* in Figure 1.4 is correct (when it is passed arguments of type `IntSet`). Does one have to go back and reverify the implementation of *inBoth* when it becomes possible to pass it arguments of type `Interval`? Since one does not have to update the code (because of message passing), it would be tiresome if one had to update the verification.

Furthermore, what does the specification of *inBoth* mean when the type `Interval` is added to the program? Consider the specification of Figure 1.6. Such a specification might be produced before the type `Interval` was contemplated. In Figure 1.6, the meaning of the operators used in the pre-condition (following **requires**) and the post-condition (following **ensures**) is expressed using functions from the specification of `IntSet`, for example $\in$, $\cap$, and "isEmpty". What does "i $\in$ s1" mean if "s1" is an `Interval`? It would be tiresome if one had to update the specification of *inBoth* when new types were added to a

[2] However, the function `inBothML` is more general than the version of *inBoth* given in Figure 1.4, because `choose`, `remove`, and `elem` do not have to be operations defined in the arguments' classes but may be arbitrary functions.

fun inBoth(s1,s2: IntSet) **returns**(i:Int)
requires ¬(isEmpty(s1 ∩ s2))
ensures (i ∈ s1) & (i ∈ s2)

Figure 1.6: Specification of the function *inBoth*.

fun inBoth[T1,T2: IntSetLikeType]
(s1:T1, s2: T2) **returns**(i:Int)
requires ¬(isEmpty(s1 ∩ s2))
ensures (i ∈ s1) & (i ∈ s2)

Figure 1.7: Traditional specification of *inBoth*.

program. Respecification would also seem to imply reverification.

To obtain the advantage of extensibility promised by object-oriented design, specification and verification techniques must be *modular* in the sense that when new type specifications are added to a design existing procedure and type specifications should not have to be changed, and when classes implementing new types are added to a program, unchanged program modules should not have to be reverified. Returning to the keys system described above, when the type `Interval` is added to the design, the existing type `IntSet` and functions such as *inBoth* should not have to be respecified, nor should the implementation of *inBoth* or other functions have to be reverified.

An obvious approach to the modular reasoning problem is to adapt standard techniques for reasoning about polymorphic program modules. The standard technique is to specify a polymorphic module by specifying the behavior of the method dictionary that the polymorphic module needs to do its work [Gut80, Page 21] [Win83, Section 4.2.3] [Gog84, Page 537]. For example, roughly following Goguen, one might specify the function *inBoth* as in Figure 1.7. The conditions that a type would have to satisfy to be a `IntSetLikeType` would be stated elsewhere, but would certainly include a specification that such a type must have operations `choose`, `remove`, and `elem` with appropriate signatures and semantics.

A minor problem with the specification of Figure 1.7 is that it is a poor match with object-oriented programming languages, because the parameterization of the specification is explicit. For example, an instantiation of *inBoth* might be written *inBoth*`[IntSet,Interval]`, which says to use the method dictionaries for `IntSet` and `Interval`. However, in an object-oriented language the method dictionaries are supplied implicitly.

The fundamental problem with the specification of Figure 1.7 is that to check that an instantiation is correct during design or verification, the actual method dictionary (i.e., the type parameter) must be statically shown to satisfy the formal's specification. However, in a program that makes use of subtype polymorphism and message passing, method dictionaries cannot, in general, be uniquely determined during design or verification. Therefore, during program verification, the exact method dictionary that will be passed at runtime cannot be determined in such a program. One might try an exhaustive case analysis by doing the verification for each possible method dictionary that could arise from the evaluation of each expression. For example, if `s1` in the expression `choose(s1)` can denote either an `IntSet` or an `Interval`, then one could instantiate the specification of `choose` with both `IntSet` and `Interval`. However, this case analysis must be extended when new subtypes are added to the program. In other words, this approach does not allow modular verification and must therefore be generalized to deal with message passing.

Conventional techniques for program verification also need to be adapted for verifying programs that use subtype polymorphism. Conventional verification techniques assume that each expression of type `T` denotes an object of type `T`. Thus the properties of the method dictionary for type `T` can be used to reason about expressions of type `T`. However, to exploit subtype polymorphism in a typed language, one must allow expressions of type `T` to denote objects of several different types (and hence different method dictionaries). The verification method must ensure that reasoning about such expressions as if they denoted instances of type `T` does not lead to invalid conclusions. Again, the traditional approach would be to use explicit polymorphism to abstract away from the changing types. But once again the problem is that the method dictionaries are not known statically, hence the use of explicit polymorphism would still lead to an exhaustive case analysis during program verification, which would have to be repeated when new types are added to the program.

The main problem then, is to design modular methods for specifying and verifying programs that use message passing and subtype polymorphism.

A related problem is how to aid the reuse of designs. In an object-oriented programming environment there tend to be many, many abstract data types (for example, [GR83] describe at least 78 types that are built-in to Smalltalk-80). Sophisticated object-oriented programming environments (e.g., [Gol84]) often provide ways to navigate or browse the code for classes that implement abstract data types. However, a designer is not primarily interested in code or subclass relationships — designers are interested in type specifications. Therefore a designer needs a concept that helps organize specifications.

1.2 Overview of a Solution

The key to solving the modular specification and verification problem for object-oriented programs is the notion of subtype relationships. If a new type of data is added to a program and the program is expected to run without changes, there must be some relationship between the behavior of existing types and the new type. For example, each object of type `Interval` behaves like some object of type `IntSet`, at least for certain implementations of `IntSet`. Hence `Interval` is a *subtype* of `IntSet`.

A modular specification and verification technique for reasoning about message passing programs can be based on the concepts of subtype relationships and nominal type, as pioneered in my dissertation [Lea89]

and further developed in [LW90]. Informally, the reasoning technique can be summarized as follows.

- One specifies the data types to be used in the program along with their subtype relationships[3].
- Procedures are specified by describing their effects on actual arguments whose types are the same as the types of the corresponding formal arguments; however, arguments whose types are subtypes of the corresponding formal argument types are permitted.
- Subtype relationships are verified to ensure that they satisfy certain semantic constraints.
- One statically associates with each expression in the program a type, called the expression's *nominal type*, with the property that an expression may only denote objects having a type that is a subtype of that expression's nominal type. (These types may be introduced solely for program verification, or they may coincide with the types of the programming language.)
- Verification that a program meets its specification is then the same as conventional verification, despite the use of message passing. That is, one reasons about expressions as if they denoted objects of their nominal types.

1.2.1 Subtyping

The key to the soundness of the method is a set of semantic requirements on subtype relationships. These behavioral constraints are like those used by programmers to reason informally about object-oriented programs. There are also some syntactic constraints on subtype relationships.

Abstract types are described by specifications that describe a set of abstract values and how the operations behave on objects with different abstract values. Such two-tiered [Win87] or abstract-model style [Jon86] specifications allow one to specify types incompletely, including types that are not intended to be implemented directly (e.g., deferred types in Eiffel [Mey88]). Such specifications also allow one to specify operations that may fail to terminate or that are nondeterministic. Because type specifications may be incomplete, they may have many different implementations with differing behavior.

More formally, a set of type specifications (e.g., including both `IntSet` and `Interval`) describes a signature Σ and a family of Σ-algebras. The signature describes the syntactic interface of the types, including the names of types and operations, the number and types of arguments for each operation, and so on. A Σ-algebra is an algebraic model of an implementation whose syntactic interface is described by Σ. Thus the meaning of a set of type specifications is a family of algebraic models with the same signature.

[3]The problem of automatically inferring subtype relationships from behavioral specifications is unsolvable, in general, since subtype relationships must satisfy certain semantic constraints. Thus the designer is required to specify the subtype relationships.

Syntactic Restrictions on Subtypes. The specificed subtype relation must satisfy certain syntactic constraints for sound reasoning. First, if one can send a message such as `choose` to a supertype object, then one must also be able to send that message to a subtype object. This prevents surprises like "message not understood." Similarly, if a function name used in specifications can be applied to a supertype object, then it should also apply to subtype objects. For example, if "isEmpty" can be applied to an `IntSet` in a specification, then one can give meaning to such specifications only if the "isEmpty" function is defined also for `Interval`. Second, if one is expecting the result type to be a subtype of `IntSet`, then the result of sending a message or using a specification function should be expected to have a type that is a subtype of `IntSet`. Such syntactic restrictions have been formalized, for example, in Reynolds's category sorted algebras [Rey80] [Rey85].

What is novel is that modularity of specifications results from the requirement that the specification functions applicable to a supertype be applicable to subtypes. Function and operation specifications are modular, because they are written as if the actual arguments had the specified types and do not explicitly mention subtypes. However, objects of subtypes of the specified types are allowed as arguments, which supports subtype polymorphism. One thinks of the meaning of a specification such as Figure 1.6 as given by using dynamic overloading of the specification function names that appear in assertions[4]. Thus if one knows that the abstract values of `iv1` and `iv2` are the intervals $[3, 27]$ and $[15, 73]$, then the result of the call *inBoth*`(iv1,iv2)` can be obtained by substituting the abstract values of the actuals for the formals in the description of the effect of *inBoth*, obtaining the formula "$(i \in [3, 27])$ & $(i \in [15, 73])$", which is interpreted using the version of $\in$ appropriate for the abstract values of intervals. Hence it is possible to discuss the testing and correctness of implementations of such specifications for all permitted arguments. Since the subtypes are not mentioned explicitly in a function or operation specification, when a new subtype is added to the program, such a specification need not be changed.

Semantic Restrictions on Subtypes. Syntactic restrictions are not enough to ensure sound, modular verification. The problem is illustrated in Figure 1.8, which illustrates static reasoning about the message passing expression `choose(s)`. Suppose that `s` is thought of as having nominal type `IntSet`, as it would be before the type `Interval` was added to the program. To conclude that the value returned by `choose`, `i`, satisfies the post-condition "$\texttt{i} \in \texttt{s}$" as specified by for the type `IntSet`, it suffices to show that the argument `s` satisfies the specified pre-condition "$\neg$ isEmpty(`s`)." This is adequate before the type `Interval` is added to the program. However, with the type `Interval` as a subtype of `IntSet`, the identifier `s`, declared to be of type `IntSet`, might denote an object s' of the subtype `Interval`. So at runtime the operation invoked is not the `choose` operation from the method dictionary associated with instances of `IntSet`, written `IntSet.choose` in the figure, but

[4]Coercing the abstract values of arguments, as in [Lea89], seems to be inferior to this approach.

instead the operation `Interval.choose`. The problem is that `Interval.choose` might not satisfy the specification used during program verification, since the `Interval.choose` operation has different pre- and post-conditions than `IntSet.choose`. Even if the pre- and post-conditions happened to be textually identical, the assertions might have different meanings for each type, since they rely on the meanings of specification functions such as "isEmpty." A solution is to require that there be an algebraic relationship, called a simulation, between the actual argument s' and the argument that was imagined during program verification (s).

Simulation relationships are, in essence, relationships among objects of different types that are preserved by message passing and by specification functions. For example, each object of type `Interval` simulates an `IntSet` object with the same elements; that is, an `Interval` with abstract value $[i, j]$ simulates an `IntSet` with abstract value $\{i, i+1, i+2, \ldots, j-1, j\}$. The preservation of simulation relationships by message passing and by assertions is called the substitution property (as in algebraic homomorphisms). For example, if `q` denotes the `Interval` $[1, 3]$ and `r` the `IntSet` $\{1, 2, 3\}$, then `q` simulates `r`. Thus by the substitution property:

$$\texttt{size(q)} \quad \text{simulates} \quad \texttt{size(r)} \tag{1.1}$$

$$\texttt{ins(q,0)} \quad \text{simulates} \quad \texttt{ins(r,0)} \tag{1.2}$$

$$2 \in \texttt{q} \quad \text{simulates} \quad 2 \in \texttt{r} \tag{1.3}$$

$$\text{isEmpty}(\texttt{q}) \quad \text{simulates} \quad \text{isEmpty}(\texttt{r}). \tag{1.4}$$

For nondeterministic operations, such as `choose`, each possible result of the `choose(q)` must simulate some possible result of `choose(r)`. Simulation is not symmetric, since `choose(r)` may have more possible results than `choose(q)`. Besides the substitution property, a simulation relation must be such that every object of a subtype simulates some object of the supertype and for built-in types such as `Bool` and `Int`, simulation is equality.

Formally, simulation relations are families of relations, one per type, between the carrier sets of two algebraic models. Each relation "simulates-as-`T`" relates elements of the carrier sets of subtypes of `T` and relates elements (i.e., instances) that cannot be distinguished using the operations applicable to instances of type `T`. A subtype may have more operations than its supertypes. For example, a type `IntTriple` may have operations `first`, `second`, and `third` and still be a subtype of `IntPair` (with operations `first` and `third`); thus, the triples $\langle 1, 2, 3\rangle$ and $\langle 1, 2, 7\rangle$ do not simulate each other as `IntTriple`s, but they do simulate each other as `IntPair`s. So that one raise one's view of objects without invalidating simulation relationships, it is required that if q simulates-as-`S` r and `S` is a subtype of `T`, then q simulates-as-`T` r.

The substitution property aids modular program verification as follows. Consider again the example of Figure 1.8. During verification, one reasons about a hypothetical object s of the type declared for the argument (`IntSet`). What happens at run-time is that the actual argument s' simulates-as-`IntSet` some hypothetical object s of type `IntSet`. Suppose the actual argument satisfies the pre-condition of `choose` specified in type `IntSet`; then since simulation is preserved by assertions, it follows that the hypothetical s also satisfies the pre-condition. So each possible hypothetical result i'' satisfies the desired post-condition "`i` $\in$ `s`." At run-time, each possible result i' must, by the substitution property for messages, simulate one of the hypothetical results, call it i. Since i satisfies the post-condition, and i' simulates i, it follows that i' satisfies the post-condition. (In this case, for the trivial reason that $i = i'$. In general such reasoning is sound provided one does not test equality ($=$) in pre- and post-conditions, except for built-in types such as `Bool` and `Int` that are assumed not to have subtypes.)

The semantic constraints on subtype relations are as follows. The idea is that for each implementation of the subtype, there must be some implementation of the supertype such that there is a simulation between the subtype objects and the supertype objects. For example, one can use an implementation of `IntSet` with a maximally nondeterministic `choose` operation to show that each `Interval` simulates some `IntSet`. It is necessary to pick an implementation of `IntSet`, because an `Interval` will not simulate an `IntSet` with the same elements in an implementation whose `choose` operation returns the greatest element. (In a given program, the implementation of `IntSet` might return the greatest element, but a program is verified against the *specification* of `IntSet`, which allows the least element to be returned.) Formally, recall that the meaning of a set of type specifications with signature Σ is a set Σ-algebras; for each of these algebraic models C, there must be some other model A such that there is a simulation from C to A.

1.2.2 Subtype versus Refinement

Subtype relationships are similar to refinement or strength [Win83, Section 5.2.1] relationships among specifications. A type `S` is *a refinement of* `T` if every implementation of the specification of `S` is an implementation of the specification of `T` . For example, a type `LeastIntSet`, which is like `IntSet` except that the `choose` operation was constrained to always return the least element of the set, would be a refinement of `IntSet`, because each implementation would also satisfy the specification of `IntSet`. However, a type `S` may be a subtype of `T`, even though `S` is not a refinement of `T`. For example, the specification of `Interval` is not a refinement of `IntSet`.

The difference between a subtype relationship and a refinement relationship is due to the distinction between class and instance operations that is made in object-oriented designs[5].

A *class* is a program module that implements an abstract data type. It describes the data structures used by instances (sometimes called instance variables) and implements the various operations of the type with procedures.

A *class operation* is typically used to create instances of a type and is not one of the operations

[5] Some object-oriented programming languages do not have classes, but are based on a notion of delegation [Lie86] [SLU89]. In such systems one can still speak of an object's instance operations (the set of messages to which it responds), even though there are no classes and class operations. Specified class operations might be implemented by functions that clone prototypes, or by instance operations of prototypes.

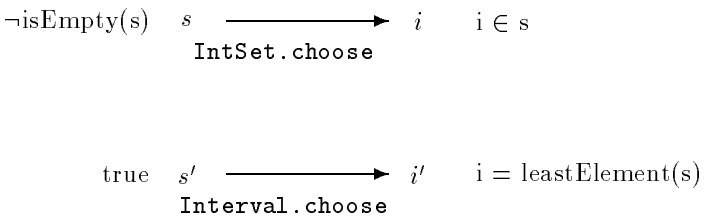

Figure 1.8: The problem with verification and message passing.

contained in instances of that type. For example, the class operation `null` of `IntSet` creates an empty integer set object and is not in the method dictionary of an `IntSet` instance. An *instance operation* is typically used to observe properties of existing instances, to make new objects from existing objects, or to change the state of an existing instance (for mutable types). For example, the instance operations `elem` of `IntSet` tests whether an integer is in the set. An instance operation is contained (conceptually) in the method dictionary of an instance. The instance operations are invoked by sending messages; whereas class operations are either invoked directly (as in C++) or by sending messages to *class objects* (as in Smalltalk-80). In a language where all operations are invoked by message passing, a class object is needed as the receiver of a message the creates an instance. For example, the class object denoted by the name `IntSet` can be used to create instances of `IntSet` by evaluating the expression `null(IntSet)`. In implementations of object-oriented programming languages such as Smalltalk-80, class objects also serve as a repository for information common to all instances of a class. (For example, class objects typically store the method dictionary shared by instances.)

For a subtype relationship only instance operations matter, since the behavior of an object, once it has been created, is only determined by its instance operations through message passing. Hence `Interval` is a subtype of `IntSet`, even though the specification of `Interval` has a class operation named `create` instead of a class operation named `null` as in `IntSet`. For a refinement relationship, both the class and instance operations matter, since an implementation of a type includes both kinds of operations. Hence `Interval` is not a refinement of `IntSet` (and vice versa). A subtype relationship is a weaker relationship than a refinement relationship, since whenever the `S` is a refinement of `T`, then `S` must be a subtype of `T`.

1.2.3 Subtype Relationships versus Subclass Relationships

Subtype relationships are not the same as subclass relationships, although most programming languages do not completely separate these notions.

A class that implements a given abstract type may be defined by stating how it differs from other classes in an object-oriented language; this mechanism is called *inheritance* or *subclassing*. If `E` is defined by inheritance from a class `D`, then `E` is called a *subclass* of `D`. Subclasses should not be confused with subtypes [Sny86b] [LTP86] [Lis88] [LaL89], as a subclass relationship is a purely implementation relationship, while a subtype relationship is a relationship among abstract data types (i.e., among specifications). In general, a subclass does not implement a subtype and a subtype need not be implemented by a subclass. For example, objects of type `Interval` can be represented by two integers; so even though `Interval` is a subtype of `IntSet`, it would not be efficient to inherit the general `IntSet` representation for the implementation of `Interval`.

C++ does not completely separate the notions of subtype and subclass, since subtype relationships are just subclass relationships that are made public (i.e., visible to clients of that type). Subclass relationships that are not used to implement subtypes can be hidden, and these hidden subclass relationships are not used in type checking. However, type checking in C++ uses subclass relationships that are not hidden as the basis for the subtype relation used in type checking. Thus the coupling of the notions of subclass and subtype limit a programmer's flexibility; for example, `Interval` would have to be implemented as a subclass of `IntSet` in order to be treated as a subtype of `IntSet` in C++.

In Trellis/Owl and Eiffel the notions of subclass and subtype are more tightly coupled, since a subtype must be implemented by a subclass and each subclass is considered to implement a subtype. When programming in such languages the programmer should only use inheritance to make subtypes (in the sense described above).

By contrast, in Smalltalk-80 or CLOS, there is no notion of type checking based on subtype relationships; hence programmers can use inheritance freely, but must enforce a disciplined use of subtypes by themselves.

1.2.4 Type Checking and Verification

With subtyping, verification for object-oriented languages is similar to verification in languages without subtype polymorphism. The main difference is that the verifier must also ensure that the specified subtype relation satisfies the semantic constraints described above. (Verification of the implementation of classes is outside the scope of this report.) Besides the semantic constraints on the specified subtype relation, type checking is necessary to ensure soundness of this

verification technique [Lea89], because it is assumed during verification that identifiers denote objects of their (statically) declared types.

Unlike a conventional programming language, a statically typed programming language that allows subtype polymorphism allows one to bind an identifier of type `T` to an object of type `S`, if `S` is a subtype of `T`. For example, an identifier of nominal type `IntSet` may denote an `Interval`, but not vice versa. In an object-oriented language there is no change of representation involved in such bindings. Environments that are created in this fashion *obey* the subtype relation in the sense that the denotation of each identifier of type `T` is an object whose type `S` is a subtype of `T`. Obedience is crucial for the soundness of verification, since if an expression of nominal type `T` may denote an object that is not an instance of a subtype of `T`, then that instance may not simulate an object of type `T`; hence reasoning about the expression using the specification of type `T` may lead to inconsistencies. A type system that ensures obedience to a subtype relation can thus aid verification by automating the verification of obedience.

1.3 Related Work

Informal treatments of object-oriented programming discuss the notion of a "protocol", which can be thought of as a specification of how an object responds to messages [GR83]. More refined treatments of this idea cast the notion of a protocol as an abstract data type specification and the notion that the protocol of one type "fits" the protocol of another type as a subtype relationship. The standard informal definition is that each object of the subtype must "behave like" some object of the supertype [Sny86b] [SCB+86].

Informal definitions of subtype relationships, like the one given above are helpful but lack the precision needed to guide designers in determining when one type is a subtype of another. For example, does the protocol of `S` "fit" the protocol of `T`, if `S` has all the instance operations of `T`, or is something more required? What `IntSet` object does a given `Interval` object behave like? Can `IntSet` be considered a subtype of `Interval`? The ability to answer such questions is necessary for formal program verification and can help guide informal reasoning.

Subtype relationships are also useful for organizing the specifications of abstract data types. Liskov has described how subtype relationships can be used during design to record decisions that refine type specifications, to localize the effects of changes to type specifications, and to group and classify types [Lis88]. LaLonde also uses subtype relationships as a means of classifying types by behavior [LTP86] [LaL89]. However, neither Liskov nor LaLonde give a formal definition of subtype relationships.

Some semi-formal specification and verification techniques appear in Meyer's book on Eiffel [Mey88, Chapter 7]. Assertions in Eiffel are written using program operations. Meyer gives specifications for operations using assertions for pre- and post-conditions. There is some discussion of loop invariants and variants, but Meyer does not give a formal logic for program verification. In chapter 11, Meyer states that a subclass should be designed to implement a subtype. The "assertion redefinition rule" states that if `r` is an operation of a class `A` and `B` is a subclass of `A`, then the pre-condition of `r` in the specification of `B` may be no stronger than the pre-condition of `r` in `A`, and the post-condition of `r` in the specification of `B` must be no weaker than the post-condition of `r` in `A` [Mey88, Page 256]. This condition is intended to ensure that the implementation of an operation in a subclass (`B`), satisfies the specification of that operation in the superclass (`A`). Meyer argues informally that this rule makes it possible to reason sensibly about programs that use message passing, although he does not give a formal verification system.

Indeed, a sound verification system cannot be constructed on Meyer's assertion language and his assertion redefinition rule. The problem is that assertions are written using program operations, but a subclass can redefine those operations; hence a subclass may satisfy the assertion redefinition rule but not have the expected behavior. The extreme of this problem occurs for deferred types; types for which one or more of the operations are not implemented (i.e., their implementation is deferred to a subclass). Consider a class `D` where all the operations are deferred. The pre- and post-conditions of the operations of `D` are written using the operations of `D`. But the operations of `D` are not implemented, so the assertions that are used to define these operations are meaningless.

Reynolds has studied partial orders on types in the setting of his category sorted algebras [Rey80] (see also [Rey85]). The semantic requirement that Reynolds imposes on the subtype relation are illustrated by the following example. Suppose `Integer` is a subtype of `Float`, a and b are objects of type `Integer`, and `to_Float` is the coercion function from `Integer` to `Float`. The coercion function `to_Float` must satisfy the *substitution property:*

$$\texttt{to\_Float}(a + b) = \texttt{to\_Float}(a) + \texttt{to\_Float}(b) \quad (1.5)$$

where the "+" on the left is `Integer` addition and the "+" on the right is `Float` addition. Requiring that the coercion and the types operations satisfy the substitution property ensures that one can reason about overloading and coercion without an exhaustive case analysis. A similar idea is found in the work of Bruce and Wegner [BW86] [BW87b].

The work of Reynolds and that of Bruce and Wegner does not allow one to directly compare type specifications: one can only compare particular algebraic models. This prohibits direct application of their work for comparing types that are incompletely specified, since for such types there may be several models with observably different behaviors. For example, the type `IntSet` is incompletely specified, as in a given implementation `choose` may return the least element of a set, the greatest element, an element chosen at random, and so on. Incomplete specifications are important in practical designs, since they leave implementation decisions open. Furthermore, the models must have carrier sets that are reduced (in the sense that distinct abstract values do not behave similarly), since otherwise there might not be a coercion function with the substitution property [Lea89, Section 5.4.3]. Finally, their models are not adequate for mutable[6]

[6] A type is *mutable* if its objects have time-varying state; such

types, types with operations that may fail to terminate, or nondeterministic operations. (On the other hand, the models used in this report are not adequate for mutable types either.)

P. America has independently developed a definition of subtype relationships [Ame89]. Like Meyer, America's definition is based on implications between pre- and post-conditions of operations. However, unlike Meyer, America does not use program operations in assertions. Instead, types are specified by describing the abstract values of their instances, and the post-condition of each program operation relates the abstract values of the arguments to the abstract value of the result. The set of abstract values of a subtype may be described differently than the set of abstract values of a supertype. Thus, for a subtype relationship, America requires a "transfer function", f, that maps the abstract values of the subtype to the abstract values of the supertype. Furthermore, for each instance operation of the supertype, it is required that

$$\text{Pre(Super)} \circ f \quad \Rightarrow \quad \text{Pre(Sub)} \tag{1.6}$$

$$\text{Post(Sub)} \quad \Rightarrow \quad \text{Post(Super)} \circ f \tag{1.7}$$

where the transfer function f is used to translate assertions of the supertype so that they apply to the abstract values of the subtype. In practice, the above requirements often mean that the transfer function must have a substitution property with respect to the program operations. As with Reynolds and Bruce and Wegner, since f must be a function, the set of abstract values must be reduced, otherwise there might not be a transfer function.

America's definition of subtyping does handle mutable types, and types that are nondeterministic. However, it does not handle aliasing, nor does it handle incompletely specified types for which there is no single set of abstract values that characterizes all implementations (i.e., for which there is no "universal model"). America's type specifications do not have class operations, there are only instance operations. The lack of class operations makes it difficult to specify types like `Interval` whose objects are created in a specific state. Because of the lack of class operations, America's notion of subtype is identical to the notion of refinement. Finally, America has not investigated modular specification and verification as discussed above.

The main line of type theoretic research on subtyping has been carried on by Luca Cardelli. His landmark paper "A Semantics of Multiple Inheritance" [Car84] showed the soundness of subtyping rules for function types, immutable records, and immutable variants. Cardelli and Mitchell give subtyping rules for immutable records that allow extension and restriction operations [CM89]. But neither of these papers nor more sophisticated types systems with the same structural ideas for subtyping (such as [Car88] and [Car89]) give subtype rules for abstract data types in general. That is, such type systems do not give general rules that can say whether `Interval` is a subtype of `IntSet` based on their specifications.

types are common in object-oriented programming. A type is *immutable* if none of its objects are mutable.

1.4 Plan of the Report

The above ideas are worked out in detail in the rest of this report. The formal details are especially needed for the treatment of examples with incomplete specifications and nondeterministic operations, for formally proving the soundness of the verification system, and for formally showing that subtyping does not lead to surprising behavior.

Algebras and simulation relations are described in Chapter 2. Simulation relations are the essence of the definition of subtyping and the proof of the soundness of verification.

Modular polymorphic type and function specifications are described in Chapter 3. The way that specifications are interpreted for arguments of a subtype of the specified argument type is described, showing that such specifications are modular.

The formal definition of subtype relations among immutable abstract types, several examples, and a more detailed comparison to related work are found in Chapter 4.

A programming language is described in Chapter 5. The language, called NOAL, has a message passing mechanism, is nondeterministic, and applicative. The nondeterminism in NOAL allows observations to be described that are similar to what one could do in a language with parallelism. Such power is important for strengthening the claims that subtyping does not lead to surprising behavior (in Chapter 7).

The verification of NOAL programs is described in Chapter 6. A Hoare-style proof system is given and its soundness is shown. The soundness of the verification system relies on the definition of subtype relations and simulation. Some modularity properties are also proved.

The properties of subtyping related to the observable behavior of programs are described in Chapter 7. The main result is that subtyping prevents surprising behavior.

A discussion of extensions and future work is found in Chapter 8.

A summary and conclusions are offered in Chapter 9. The conclusions include lessons for programmers and language designers.

The notation and definitions used in the report are summarized in Appendix A. The built-in types of the specification language are described in Appendix B. The semantics of NOAL functions are described in Appendix C.

As may be seen from the above, several simplifying assumptions are made in this report. Only designs with immutable types are considered, and verification is only considered for an applicative object-oriented language. Thus the report represents a second step (after [Lea89]) towards general reasoning techniques for programs that use message passing and subtype polymorphism.

Chapter 2

Algebraic Models and Simulation Relations

In this chapter the semantics of sets of type specifications, sets of algebraic models, and simulation relations between these models are discussed. Both concepts are crucial to the definition of subtype relations and to the treatment of specification and verification. Algebras are presented first, followed by simulation relations.

2.1 Algebraic Models

The algebras defined below are an extension of the usual algebraic structures found in the study of equational logic or algebraic specifications [EM85]. As such an algebra includes a carrier set and a set of *specification functions*; to these are added a set of *program operations*. The specification functions are used to generate the carrier sets and in the evaluation of the assertions used in specifications [Win83, Chapter 2], while the program operations are used by programs for computation. The specification functions cannot be invoked by programs, and the program operations cannot be used in specifications.

The program operations of an algebra are abstractions of the procedures of the classes that implement abstract types in object-oriented programs. To model nondeterministic procedures, the program operations of an algebra are set-valued functions; that is, a program operation returns the set of the possible results of the corresponding procedure [Nip86] [Nip87]. The special value $\perp$ is used to model procedure calls that do not halt or that encounter run-time errors. So a procedure that might either return 1 or never halt on some argument q would be modeled by a program operation that, when called with q, has $\{1, \perp\}$ as its set of possible results.

To more closely model the procedures of a class, the program operations of an algebra may be polymorphic. The treatment of polymorphism is based on techniques from Reynolds's category sorted algebras [Rey80]. Message passing is thus modeled by simply invoking a program operation.

The specification functions have no counterpart in the classes that implement abstract types. Rather the specification functions are models of the functions used to precisely specify the abstract values of a type. Such specification in a *trait*. Algebraic equational specifications, called *traits*, are used to specify abstract values; these are written in the Larch Shared Language [GH86b]. To ensure that the assertions used in specifications have meaning when subtyping is used, specification functions are also polymorphic.

In an algebra there is no separate representation for abstract values and objects. That is, the objects of a type are identified with their abstract values. (This is adequate for immutable types, which are the only ones considered in this report.) So each type symbol is also a *sort* symbol. Sorts play the same role in traits as types do in programs. Only types can be used in programs, but in general one may need auxiliary sorts for convenience in specification.

There is a small set of types in an algebra called the *visible types*. These types, such as `Bool` and `Int`, are used to define observations.

2.1.1 Signatures

A set of type specifications describes a set of algebras. The specifications describe the set of traits used, which determine the names of the sorts and the specification functions, as well as the signatures of the specification functions. The specifications also describe each type and its program operations, and hence determines a set of type names, and a set of program operation symbols. The set of visible types is fixed by convention, but is included in signatures for convenience.

Type specifications also describe a binary relation on sort symbols, $\leq$, which is the *presumed subtype relation* for the specification. Type specifications also determine the nominal signatures for each of the program operations of each type. From the nominal signature information is derived a function that predicts an upper bound (using $\leq$) of the result type of a program operation or specification function [Rey80].

Definition 2.1.1 (signature). A *signature*

$$\Sigma = \left(\begin{array}{c} SORTS, TYPES, V, \leq, \\ SFUNS, POPS, ResSort \end{array} \right)$$

consists of:

- Sets of sort, type, and visible type symbols, such that $V \subseteq TYPES \subseteq SORTS$ and V is nonempty.
- A binary relation, $\leq$, which is a preorder[1] on $SORTS$, such that for all visible types $\mathtt{T} \in V$, if $\mathtt{S} \leq \mathtt{T}$ then $\mathtt{S} = \mathtt{T}$.
- Disjoint sets, $SFUNS$ of specification function symbols, and $POPS$ of program operation symbols. Their union, the set of all operation symbols, is denoted OPS:

$$OPS \stackrel{\text{def}}{=} SFUNS \cup POPS. \qquad (2.1)$$

[1] A *preorder* is a relation that is reflexive and transitive. It is not necessarily antisymmetric.

- A partial function $ResSort: OPS, SORTS^* \rightarrow SORTS$, which returns an upper bound on the result sort of a specification function or program operation symbol applied to a tuple of arguments with the given sorts. Following Reynolds [Rey80, Page 217], *ResSort* must be monotone in the following sense: for all $\mathtt{g} \in OPS$, and for all tuples of sorts $\vec{\mathtt{S}} \leq \vec{\mathtt{T}}$, if $ResSort(\mathtt{g}, \vec{\mathtt{T}})$ is defined, then so is $ResSort(\mathtt{g}, \vec{\mathtt{S}})$, and furthermore $ResSort(\mathtt{g}, \vec{\mathtt{S}}) \leq ResSort(\mathtt{g}, \vec{\mathtt{T}})$.

The restriction on $\leq$ ensures that there can be no subtypes of a visible type. This restriction is reasonable, since only visible types can appear as the output of programs an object of some other type cannot behave quite like an object of a visible type.

The restrictions on *ResSort* ensure that if an operation can be applied to arguments of a given type, then the same operation can be applied to arguments of any presumed subtype of that type. This ensures, for example, that a presumed subtype has all the generic operations defined for its presumed supertypes, as one would expect. More restrictive is the requirement that the specification functions also apply to presumed subtypes. Thus if one wishes to specify a subtype, one must define all the trait functions of the presumed supertype on the abstract values. In [Lea89], there was no such restriction, but its lack lead to strong restrictions on the kinds of assertions that one could write in a specification.

An example signature, is given in Figure 2.1; this signature is derived (as described in Chapter 3) from the specification that combines the types `IntSet` (Figure 1.1) and `Interval` (Figure 1.2). To permit more flexibility than would be allowed by a strict application of the rule that specification function symbols cannot be used as program operation symbols, different fonts are used for each kind of symbol. The specification function symbols are written in a Roman font (f), and program operation symbols appear in typewriter font (g). Some specification functions symbols contain sharp signs (#) as place holders for arguments — these allow the use of prefix and infix syntax in assertions. The sorts include, `Card`, `IntSet`, `Interval`, `IntSetClass`, `IntervalClass`, and sorts for the visible types (`Bool` and `BoolClass`, `Int`, and so on). All of these sorts are also types, except `Card` (cardinal numbers), which is used only as an auxiliary sort in specifications (and thus cannot be used in programs). The types `Bool`, `Int`, `BoolStream`, and `IntStream` are visible types. The relation $\leq$ relates `Interval` to `IntSet` and each sort to itself. "IntSet" is also the name of a specification function (the specification function that returns the abstract value of the class object for the type `IntSet`), as are versions of the other type names that are not class types. Specification functions must be defined for all arguments of a subtype; hence "insert" may take either an `IntSet` or an `Interval` as its first argument. The program operations include nullary operations named `IntSet` and `Interval`, class operations named `null` and `create`, and the specified instance operations, as well as operations for the visible types.

The notion of subsignature is important in the study of modular verification.

$$
\begin{array}{lcl}
SORTS & = & \{\mathtt{Card}, \mathtt{IntSet}, \mathtt{IntSetClass}, \\
 & & \mathtt{Interval}, \mathtt{IntervalClass}, \\
 & & \mathtt{Bool}, \ldots\} \\
TYPES & = & \{\mathtt{IntSet}, \mathtt{IntSetClass}, \\
 & & \mathtt{Interval}, \mathtt{IntervalClass}, \\
 & & \mathtt{Bool}, \ldots\} \\
V & = & \{\mathtt{Bool}, \mathtt{Int}, \\
 & & \mathtt{BoolStream}, \mathtt{IntStream}\}
\end{array}
$$

Presumed Subtype Relation ($\leq$)

$$
\begin{array}{lcl}
\mathtt{Interval} & \leq & \mathtt{IntSet} \\
\mathtt{Interval} & \leq & \mathtt{Interval} \\
\mathtt{IntSet} & \leq & \mathtt{IntSet} \\
 & \ldots &
\end{array}
$$

ResSort for Program Operations

$$
\begin{array}{lcl}
\mathtt{IntSet}, \langle\rangle & \mapsto & \mathtt{IntSetClass} \\
\mathtt{Interval}, \langle\rangle & \mapsto & \mathtt{IntervalClass} \\
\mathtt{null}, \langle \mathtt{IntSetClass} \rangle & \mapsto & \mathtt{IntSet} \\
\mathtt{create}, \langle \mathtt{IntervalClass}, \mathtt{Int}, \mathtt{Int} \rangle & & \\
 & \mapsto & \mathtt{IntSet} \\
\mathtt{ins}, \langle \mathtt{IntSet}, \mathtt{Int} \rangle & \mapsto & \mathtt{IntSet} \\
\mathtt{ins}, \langle \mathtt{Interval}, \mathtt{Int} \rangle & \mapsto & \mathtt{IntSet} \\
\mathtt{elem}, \langle \mathtt{IntSet}, \mathtt{Int} \rangle & \mapsto & \mathtt{Bool} \\
\mathtt{elem}, \langle \mathtt{Interval}, \mathtt{Int} \rangle & \mapsto & \mathtt{Bool} \\
\mathtt{choose}, \langle \mathtt{IntSet} \rangle & \mapsto & \mathtt{Int} \\
\mathtt{choose}, \langle \mathtt{Interval} \rangle & \mapsto & \mathtt{Int} \\
\mathtt{size}, \langle \mathtt{IntSet} \rangle & \mapsto & \mathtt{Int} \\
\mathtt{size}, \langle \mathtt{Interval} \rangle & \mapsto & \mathtt{Int} \\
\mathtt{remove}, \langle \mathtt{IntSet}, \mathtt{Int} \rangle & \mapsto & \mathtt{IntSet} \\
\mathtt{remove}, \langle \mathtt{Interval}, \mathtt{Int} \rangle & \mapsto & \mathtt{IntSet} \\
\mathtt{Bool} & \mapsto & \mathtt{BoolClass} \\
 & \ldots &
\end{array}
$$

Figure 2.1: Signature for the specification II of `IntSet` and `Interval`, first part.

ResSort for Specification Functions

IntSet, $\langle\rangle$	$\mapsto$	`IntSetClass`
$\{\}, \langle\rangle$	$\mapsto$	`IntSet`
insert, $\langle$`IntSet`, `Int`$\rangle$	$\mapsto$	`IntSet`
insert, $\langle$`Interval`, `Int`$\rangle$	$\mapsto$	`IntSet`
size, $\langle$`IntSet`$\rangle$	$\mapsto$	`Card`
size, $\langle$`Interval`$\rangle$	$\mapsto$	`Card`
toInt, $\langle$`Card`$\rangle$	$\mapsto$	`Int`
$\# \in \#, \langle$`Int`, `IntSet`$\rangle$	$\mapsto$	`Bool`
$\# \in \#, \langle$`Int`, `Interval`$\rangle$	$\mapsto$	`Bool`
delete, $\langle$`IntSet`, `Int`$\rangle$	$\mapsto$	`IntSet`
delete, $\langle$`Interval`, `Int`$\rangle$	$\mapsto$	`IntSet`
$\# == \#, \langle$`IntSet`, `IntSet`$\rangle$	$\mapsto$	`IntSet`
$\# == \#, \langle$`IntSet`, `Interval`$\rangle$	$\mapsto$	`IntSet`
$\# == \#, \langle$`Interval`, `IntSet`$\rangle$	$\mapsto$	`IntSet`
$\# == \#, \langle$`Interval`, `Interval`$\rangle$	$\mapsto$	`IntSet`
$\# \cap \#, \langle$`IntSet`, `IntSet`$\rangle$	$\mapsto$	`IntSet`
$\# \cap \#, \langle$`IntSet`, `Interval`$\rangle$	$\mapsto$	`IntSet`
$\# \cap \#, \langle$`Interval`, `IntSet`$\rangle$	$\mapsto$	`IntSet`
$\# \cap \#, \langle$`Interval`, `Interval`$\rangle$	$\mapsto$	`IntSet`
$\{\#\}, \langle$`Int`$\rangle$	$\mapsto$	`IntSet`
$\# \cup \#, \langle$`IntSet`, `IntSet`$\rangle$	$\mapsto$	`IntSet`
$\# \cup \#, \langle$`IntSet`, `Interval`$\rangle$	$\mapsto$	`IntSet`
$\# \cup \#, \langle$`Interval`, `IntSet`$\rangle$	$\mapsto$	`IntSet`
$\# \cup \#, \langle$`Interval`, `Interval`$\rangle$	$\mapsto$	`IntSet`
isEmpty, $\langle$`IntSet`$\rangle$	$\mapsto$	`Bool`
isEmpty, $\langle$`Interval`$\rangle$	$\mapsto$	`Bool`
Interval, $\langle\rangle$	$\mapsto$	`IntervalClass`
$[\#, \#], \langle$`Int`, `Int`$\rangle$	$\mapsto$	`Interval`
toSet, $\langle$`Interval`$\rangle$	$\mapsto$	`IntSet`
leastElement, $\langle$`Interval`$\rangle$	$\mapsto$	`Int`
greatestElement, $\langle$`Interval`$\rangle$	$\mapsto$	`Int`
Bool, $\langle\rangle$	$\mapsto$	`BoolClass`
	...	

Figure 2.2: Signature for the specification II of `IntSet` and `Interval`, last part.

Definition 2.1.2 (subsignature). A signature Σ' is a *subsignature* of Σ if $SORTS' \subseteq SORTS$, $TYPES' \subseteq TYPES$, $V' \subseteq V$, $SFUNS' \subseteq SFUNS$, $POPS' \subseteq POPS$, $\leq'$ is the restriction of $\leq$ to $SORTS'$, $ResSort(OPS' \times (SORTS')^*) \subseteq SORTS'$, $ResSort'$ is the restriction of $ResSort$ to $OPS' \times (SORTS')^*$, and if for all sorts $\mathtt{S}', \mathtt{T}' \in SORTS'$, if there is a sort $\mathtt{U}'$ that is the least upper bound of $\mathtt{S}'$ and $\mathtt{T}'$ in $\leq'$, then for all sorts $\mathtt{S}, \mathtt{T} \in SORTS$, whenever $\mathtt{S} \leq \mathtt{S}'$ and $\mathtt{T} \leq \mathtt{T}'$, then the least upper bound of $\mathtt{S}$ and $\mathtt{T}$ exists and is a sort $\mathtt{U} \leq \mathtt{U}'$.

2.1.2 Algebras

Algebras reflect the two-tiered structure of specifications (traits and specifications of program operations) in that an algebra is built on a model of the traits.

A *trait signature*, $\sigma = (SORTS, SFUNS, ResSort)$, where $SORTS$, $SFUNS$ and $ResSort$ are as above. A *σ-trait structure* is an algebraic structure, $A = (|A|, SFUNS^A)$, having trait signature σ, hence it is what is (usually) called an algebra in the study of equational logic or algebraic specifications. That is, a trait structure consists of a carrier set $|A|$, and a set of specification functions $SFUNS^A$.

For each sort $\mathtt{T}$ in $SORTS$ a trait structure A has a set, $\mathtt{T}^A$, called the *carrier set of* $\mathtt{T}$. So the *carrier set of trait structure* A, written $|A|$, is a $SORTS$-indexed family of sets:

$$|A| \stackrel{\text{def}}{=} \{\mathtt{T}^A \mid \mathtt{T} \in SORTS\}. \tag{2.2}$$

(An *I-indexed set* is a surjective function from an index set I to the elements of a set. For example, $|A|$ is a $SORTS$-indexed set where each element is a set.)

A *specification function* is function that takes a tuple of zero or more elements of a carrier set and returns an element of the carrier set.

The set of sorted specification functions of a trait structure must not contradict the signature; that is, for each specification function symbol $\mathrm{f} \in SFUNS$, and for each tuple of sorts $\vec{\mathtt{S}}$, if there is some sort $\mathtt{T}$ such that $ResSort(\mathrm{f}, \vec{\mathtt{S}}) = \mathtt{T}$, then for each tuple $\vec{q} \in \vec{\mathtt{S}}^A$, $\mathrm{f}^A(\vec{q}) \in \mathtt{T}^A$.

An algebra is formed from a trait structure by making $\perp$ an element of the carrier set of each sort, extending the specification functions to this larger carrier set, and adding a set of program operations.

The *carrier set of a Σ-algebra* A, written $|A|$, consists of the carrier set of its trait structure, together with $\perp$, which is made an element of each sort's carrier set. The family of carrier sets for the types is written $TYPES^A$.

An element of a carrier set is called *proper* if it is not $\perp$.

The phrase "q has sort $\mathtt{T}$" means that $q \in \mathtt{T}^A$; furthermore, if $\mathtt{T}$ is also type, the phrases "q has type $\mathtt{T}$" and "q is an instance of type $\mathtt{T}$" mean the same thing. An element of a carrier set, often written o, q, or r, is also called an *object*.

The *family of sorted specification functions* of a Σ-algebra A, written $SFUNS^A$, consists of a set of sorted specification functions for a trait structure, extended

so that they do not interact with $\perp$. That is, in addition to the constraints on the specification functions described above, for each $\mathrm{f} \in SFUNS$, the specification function f^A has $\perp$ as a result if and only if one of its arguments is $\perp$.

The *family of program operations* of a Σ-algebra, A, written $POPS^A$ is a $POPS$-indexed family of set-valued functions that do not contradict the sort information in the signature. That is, for each $\mathtt{g} \in POPS$, for each tuple of types $\vec{\mathtt{S}}$, and for each tuple $\vec{q} \in \vec{\mathtt{S}}^A$, if $ResSort(\mathtt{g}, \vec{\mathtt{S}}) = \mathtt{U}$, then $\mathtt{g}^A(\vec{q})$ is a nonempty set and for each $r \in \mathtt{g}^A(\vec{q})$, there is some type $\mathtt{T} \leq \mathtt{U}$ such that r has type $\mathtt{T}$. Notice that the return type of an operation is more loosely constrained than the return sort of a specification function.

As above, vector notation is often used for tuples. For example, the notation $\vec{q}$ stands for a n-tuple $\langle q_1, \ldots, q_n \rangle$, where $n \geq 0$. The tuple $\vec{q}$ has type $\vec{\mathtt{S}}$, written $\vec{q} \in \vec{\mathtt{S}}^A$, if each q_i has type $\mathtt{S}_i$. The notation $\vec{\mathtt{S}}^A$ stands for ${\mathtt{S}_1}^A \times \cdots \times {\mathtt{S}_n}^A$, but if $\vec{\mathtt{S}} = \langle\rangle$, then $\vec{\mathtt{S}}^A \stackrel{\text{def}}{=} \{\langle\rangle\}$. The same notation is also used for sorts.

Notice that the definition of the program operations of an algebra is more general than the usual definition of an operation, since the program operations are polymorphic and possibly nondeterministic. The operations even generalize Reynolds' generic operators [Rey80], in that the result returned may have some type other than the result type given by *ResSort*.

A *possible result* is simply an element of the carrier set of an algebra. program operations can also be thought of as binary relations, where the set of possible results of an application of a binary relation f to a tuple of arguments $\vec{q}$ is

$$f(\vec{q}) \stackrel{\text{def}}{=} \{r \mid (\vec{q}, r) \in f\}. \tag{2.3}$$

The above is summarized in the following definition.

Definition 2.1.3 (algebra). Let Σ be a signature. An Σ-*algebra* $A = (|A|, SFUNS^A, POPS^A)$, consists of:

- a carrier set, $|A|$,
- a family, $SFUNS^A$, of sorted specification functions, and
- a family, $POPS^A$, of program operations.

A Σ-algebra has signature Σ.

The program operations of algebras can be non-strict, in contrast with the operations of Nipkow's algebras. An operation is *strict* if whenever one of its arguments is $\perp$, then the only possible result is $\perp$. Specification functions are strict. However, non-strict program operations are useful for modeling types with lazy evaluation, such as streams.

Operations and algebras can be classified as follows [Nip87, Page 9]. An operation is *total* if whenever all its arguments are proper (i.e., not $\perp$), then no possible result is $\perp$. Thus each specification function is total. An operation that is not total is *partial*. If an application of an operation has a single possible result, it is *deterministic*. An operation is deterministic if all applications of that operation are deterministic. Thus each specification function is deterministic. An operation that is not deterministic is *nondeterministic*. An algebra is total or deterministic if all its operations have that property. An algebra is partial or nondeterministic if some of its operations are partial or nondeterministic.

An example $SIG(II)$-algebra, where $SIG(II)$ is the signature of Figure 2.1, is presented in Figure 2.3. The figure first defines the carrier sets for each type, then the specification functions, and finally the possible results of the program operations. The carrier set for the type `IntSet` contains $\perp$, and finite sets of integers. The only proper element of the carrier set for `IntSetClass` is *IntSet*, which is used as the result of the nullary program operation `IntSet`. The description of specification functions and operations is abbreviated by the following conventions.

- A variable such as i only stands for a proper element of the appropriate carrier set, never for $\perp$.
- The only omitted cases involve $\perp$ as an argument, and for these the only possible result is $\perp$.
- The specification functions are used to define the operations.

Note that the specification functions are thought of as a collection of separate functions, some of which share the same name, while the operations are thought of as monolithic. Thus each operation has a single definition that applies to all elements of the appropriate carrier sets. Also note that the `choose` operation of B is defined on nonempty arguments of type `IntSet` by to have the entire set argument as its set of possible results, but is deterministic for arguments of type `Interval`. Also, if `choose` is applied to an empty `IntSet`, then the possible results are the entire carrier set of `Int`, including $\perp$.

Algebras that are abstractions of implementations of abstract types written in a given programming language all have the same set of visible (i.e., built-in) types. Alternatively, one can think of the visible types as fixed by one's specification language. In either case, one can use the visible types to define behaviors only if the meaning of a visible type is the same in each algebra. To state this restriction precisely requires the notion of the reduct of an algebra.

Let Σ' be a subsignature of Σ. Let $SORTS'$, $SFUNS'$, and $POPS'$ be the sets of sort, specification function, and program operation symbols of Σ'. Let A be a Σ-algebra. Then the Σ'-*reduct of* A is the algebra

$$A_{(\Sigma')} \stackrel{\text{def}}{=} \begin{pmatrix} \{\mathtt{T}^A \mid \mathtt{T} \in SORTS'\}, \\ \{\mathrm{f}^A \mid \mathrm{f} \in SFUNS'\}, \\ \{\mathtt{g}^A \mid \mathtt{g} \in POPS'\} \end{pmatrix} \tag{2.4}$$

[EM85, Section 6.8]. That is, $A_{(\Sigma')}$ has as its carrier sets the carrier sets of the sorts in A that appear in Σ', and as its specification functions and program operations those named in Σ'.

Assume that there is some fixed signature ΣB and some fixed ΣB-algebra, B, that defines the visible types. Then the assumption that all implementations have the same visible types amounts to an assumption that all signatures considered will have ΣB as a subsignature and all algebras will have B as their ΣB-reduct.

Carrier Sets

$$
\begin{array}{lcl}
\texttt{IntSet}^B & \stackrel{\text{def}}{=} & \{\bot\} \cup \text{“finite sets of proper elements of } \texttt{Int}^B\text{”} \\
\texttt{IntSetClass}^B & \stackrel{\text{def}}{=} & \{\bot, \mathit{IntSet}\} \\
\texttt{Interval}^B & \stackrel{\text{def}}{=} & \{\bot\} \cup \{[x,y] \mid x, y \in \texttt{Int}^B, x \neq \bot, y \neq \bot, x \leq y\} \\
\texttt{IntervalClass}^B & \stackrel{\text{def}}{=} & \{\bot, \mathit{Interval}\} \\
\texttt{Bool}^B & \stackrel{\text{def}}{=} & \{\bot, \mathit{true}, \mathit{false}\} \\
 & \ldots &
\end{array}
$$

Specification Functions, first part

$$
\begin{array}{lcl}
\text{IntSet}^B() & \stackrel{\text{def}}{=} & \mathit{IntSet} \\
\{\}^B() & \stackrel{\text{def}}{=} & \{\} \\
\text{insert}^B(\{\}, i) & \stackrel{\text{def}}{=} & \{i\} \\
\text{insert}^B(\{i_1, \ldots, i_n\}, i) & \stackrel{\text{def}}{=} & \{i_1, \ldots, i_n\} \cup \{i\} \\
\text{insert}^B([x,y], i) & \stackrel{\text{def}}{=} & \text{toSet}^B([x,y]) \cup \{i\} \\
\text{size}^B(\{\}) & \stackrel{\text{def}}{=} & 0 \\
\text{size}^B(\{i_1, \ldots, i_n\}) & \stackrel{\text{def}}{=} & 1 + \text{size}^B(\{i_1, \ldots, i_{n-1}\}) \\
\text{size}^B([x,y]) & \stackrel{\text{def}}{=} & \text{size}^B(\text{toSet}^B(v)) \\
\text{toInt}^B(0) & \stackrel{\text{def}}{=} & 0 \\
\text{toInt}^B(\mathit{succ}(c)) & \stackrel{\text{def}}{=} & 1 + \text{toInt}^B(c) \\
\# \in \#^B(i, \{\}) & \stackrel{\text{def}}{=} & \mathit{false} \\
\# \in \#^B(i, \{i_1, \ldots, i_n\}) & \stackrel{\text{def}}{=} & i \in \{i_1, \ldots, i_n\} \\
\# \in \#^B(i, [x,y]) & \stackrel{\text{def}}{=} & x \leq i \wedge i \leq y \\
\text{delete}^B(\{\}, i) & \stackrel{\text{def}}{=} & \{\} \\
\text{delete}^B(\{i_1, \ldots, i_n\}, i) & \stackrel{\text{def}}{=} & \{i_1, \ldots, i_n\} \setminus \{i\} \\
\text{delete}^B([x,y], i) & \stackrel{\text{def}}{=} & \text{toSet}^B([x,y]) \setminus \{i\} \\
\# == \#^B(\{\}, \{\}) & \stackrel{\text{def}}{=} & \mathit{true} \\
\# == \#^B(\{\}, \{i_1, \ldots, i_n\}) & \stackrel{\text{def}}{=} & \mathit{false} \\
\# == \#^B(\{i_1, \ldots, i_n\}, \{\}) & \stackrel{\text{def}}{=} & \mathit{false} \\
\# == \#^B(\{i_1, \ldots, i_n\}, \{j_1, \ldots, j_m\}) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{ll} \mathit{true} & \text{if } \{i_1, \ldots, i_n\} \subseteq \{j_1, \ldots, j_m\} \\ & \text{and } \{j_1, \ldots, j_m\} \subseteq \{i_1, \ldots, i_n\} \\ \mathit{false} & \text{otherwise} \end{array} \right. \\
\# == \#^B(\{i_1, \ldots, i_n\}, [x,y]) & \stackrel{\text{def}}{=} & \# == \#^B(\{i_1, \ldots, i_n\}, \text{toSet}^B([x,y])) \\
\# == \#^B([x,y], \{i_1, \ldots, i_n\}) & \stackrel{\text{def}}{=} & \# == \#^B(\{i_1, \ldots, i_n\}, \text{toSet}^B([x,y])) \\
\# == \#^B([x_1,y_1], [x_2,y_2]) & \stackrel{\text{def}}{=} & \# == \#^B(\text{toSet}^B([x_1,y_1]), \text{toSet}^B([x_2,y_2])) \\
\# \cap \#^B(\{\}, s_2) & \stackrel{\text{def}}{=} & \{\} \\
\# \cap \#^B(s_1, \{\}) & \stackrel{\text{def}}{=} & \{\} \\
\# \cap \#^B(\{i_1, \ldots, i_n\}, \{j_1, \ldots, j_n\}) & \stackrel{\text{def}}{=} & \{i_1, \ldots, i_n\} \cap \{j_1, \ldots, j_n\} \\
\# \cap \#^B(\{i_1, \ldots, i_n\}, [x,y]) & \stackrel{\text{def}}{=} & \{i_1, \ldots, i_n\} \cap \text{toSet}^B([x,y]) \\
\# \cap \#^B([x,y], \{i_1, \ldots, i_n\}) & \stackrel{\text{def}}{=} & \{i_1, \ldots, i_n\} \cap \text{toSet}^B([x,y])
\end{array}
$$

Figure 2.3: An algebra B for the specification II, including `IntSet` and `Interval`, first part.

Specification Functions, last part

$$
\begin{array}{lcl}
\# \cap \#^B([x_1,y_1],[x_2,y_2]) & \stackrel{\text{def}}{=} & \text{toSet}^B([x_1,y_1]) \cap \text{toSet}^B([x_2,y_2]) \\
\{\#\}^B(i) & \stackrel{\text{def}}{=} & \{i\} \\
\# \cup \#^B(\{\},s_2) & \stackrel{\text{def}}{=} & s_2 \\
\# \cup \#^B(s_1,\{\}) & \stackrel{\text{def}}{=} & s_1 \\
\# \cup \#^B(\{i_1,\ldots,i_n\},\{j_1,\ldots,j_n\}) & \stackrel{\text{def}}{=} & \{i_1,\ldots,i_n\} \cup \{j_1,\ldots,j_n\} \\
\# \cup \#^B(\{i_1,\ldots,i_n\},[x,y]) & \stackrel{\text{def}}{=} & \{i_1,\ldots,i_n\} \cup \text{toSet}^B([x,y]) \\
\# \cup \#^B([x,y],\{i_1,\ldots,i_n\}) & \stackrel{\text{def}}{=} & \{i_1,\ldots,i_n\} \cup \text{toSet}^B([x,y]) \\
\# \cup \#^B([x_1,y_1],[x_2,y_2]) & \stackrel{\text{def}}{=} & \text{toSet}^B([x_1,y_1]) \cup \text{toSet}^B([x_2,y_2]) \\
\text{isEmpty}^B(\{\}) & \stackrel{\text{def}}{=} & \mathit{true} \\
\text{isEmpty}^B(\{i_1,\ldots,i_n\}) & \stackrel{\text{def}}{=} & \mathit{false} \\
\text{isEmpty}^B([x,y]) & \stackrel{\text{def}}{=} & x \leq y \\
 & & \\
\text{Interval}^B([x,y]) & \stackrel{\text{def}}{=} & \mathit{Interval} \\
[\#,\#]^B(x,y) & \stackrel{\text{def}}{=} & \begin{cases} [x,y] & \text{if } x \leq y \\ [x,x] & \text{if } x > y \end{cases} \\
\text{toSet}^B([x,y]) & \stackrel{\text{def}}{=} & \begin{cases} \{x\} & \text{if } x = y \\ \{x\} \cup \text{toSet}^B([x+1,y]) & \text{if } x < y \\ \{x\} & \text{otherwise} \end{cases} \\
\text{leastElement}^B([x,y]) & \stackrel{\text{def}}{=} & x \\
\text{greatestElement}^B([x,y]) & \stackrel{\text{def}}{=} & y \\
 & & \\
\text{Bool}^B() & \stackrel{\text{def}}{=} & \mathit{Bool} \\
 & \ldots &
\end{array}
$$

Program Operations

$$
\begin{array}{lcl}
\texttt{IntSet}^B() & \stackrel{\text{def}}{=} & \{\mathit{IntSet}\} \\
\texttt{Interval}^B() & \stackrel{\text{def}}{=} & \{\mathit{Interval}\} \\
\texttt{null}^B(\mathit{IntSet}) & \stackrel{\text{def}}{=} & \{\{\}\} \\
\texttt{create}^B(\mathit{Interval},x,y) & \stackrel{\text{def}}{=} & \{[\#,\#]^B(x,y)\} \\
\texttt{ins}^B(s,i) & \stackrel{\text{def}}{=} & \begin{cases} \{[x,y]\} & \text{if } s \in \texttt{Interval}^B,\ s=[x,y],\ x \leq i \leq y \\ \{[i,y]\} & \text{if } s \in \texttt{Interval}^B,\ s=[x,y],\ i = x-1 \\ \{[x,i]\} & \text{if } s \in \texttt{Interval}^B,\ s=[x,y],\ i = y+1 \\ \{\text{insert}^B(s,i)\} & \text{otherwise.} \end{cases} \\
\texttt{elem}^B(s,i) & \stackrel{\text{def}}{=} & \{\# \in \#^B(s,i)\} \\
\texttt{choose}^B(s) & \stackrel{\text{def}}{=} & \begin{cases} \texttt{Int}^B & \text{if } s \in \texttt{IntSet}^B,\ s=\{\} \\ s & \text{if } s \in \texttt{IntSet}^B,\ s \neq \{\} \\ \{x\} & \text{if } s \in \texttt{Interval}^B,\ s=[x,y] \end{cases} \\
\texttt{size}^B(s) & \stackrel{\text{def}}{=} & \{\text{toInt}^B(\text{size}^B(s))\} \\
\texttt{remove}^B(s,i) & \stackrel{\text{def}}{=} & \{\text{delete}^B(s,i)\} \\
 & & \\
\texttt{Bool}^B() & \stackrel{\text{def}}{=} & \{\mathit{Bool}\} \\
 & \ldots &
\end{array}
$$

Figure 2.4: An algebra B for the specification II, including `IntSet` and `Interval`, last part.

2.2 Simulation Relations

How can one prove that one object "behaves like" another object? An algebraic technique is to model the notion of "behaves like" with a relation that satisfies the substitution property and is the identity on objects of visible type. The statement that q behaves like r is expressed as the relationship $q\ R\ r$. The substitution property is used to show that in all contexts $P(\cdot)$, if $q\ R\ r$, then $P(q)\ R\ P(r)$. The notion of "behavior" concerns a subset of all contexts, namely programs. Programs by their nature can only produce objects of visible type such as `Bool` or `Int`. Such types are called *visible* types. By ensuring that the relation R is the identity on the visible types, one ensures that if $q\ R\ r$, then q behaves like r with respect to all programs.

The above story is slightly complicated if one's programs are typed. Then one cannot apply arbitrary programs to objects, but only those programs that type check. The situation is further complicated when one takes subtyping into account, because then the observations that can be applied to an object depend on what type one assumes for the object. For example, one can apply more program operations to a triple than to a pair, hence whether two triples behave like each other depends on whether one observes them as pairs or triples. Therefore, simulation relations are families of relations that have one binary relation per sort. At each type the relation may relate elements of presumed subtypes of that type. For example, $\mathcal{R}_{\mathtt{T}}$ may relate two instances of type `S` (provided $\mathtt{S} \leq \mathtt{T}$), an instance of `S` to an instance of type `T`, or vice versa.

Definition 2.2.1 (simulation). Let C and A be Σ-algebras. A $SORTS$-indexed family

$$\mathcal{R} = \left\{ \mathcal{R}_{\mathtt{T}} \subseteq ((\bigcup_{\mathtt{S} \leq \mathtt{T}} \mathtt{S}^C) \times (\bigcup_{\mathtt{S} \leq \mathtt{T}} \mathtt{S}^A)) \mid \mathtt{T} \in SORTS \right\},$$

is a Σ-*simulation relation between* C *and* A, if and only if the following properties hold:

Substitution property: for all sorts `T`, for all tuples of sorts $\vec{\mathtt{S}}$, $\vec{\mathtt{U}}$, and $\vec{\mathtt{V}}$ such that $\vec{\mathtt{U}} \leq \vec{\mathtt{S}}$ and $\vec{\mathtt{V}} \leq \vec{\mathtt{S}}$, for all tuples $\vec{q} \in \vec{\mathtt{U}}^C$, for all tuples $\vec{r} \in \vec{\mathtt{V}}^A$ such that $\vec{q}\,\mathcal{R}_{\vec{\mathtt{S}}}\,\vec{r}$, the following hold:

- for all specification function symbols $\mathrm{f} \in SFUNS$, such that $ResSort(\mathrm{f}, \vec{S}) = \mathtt{T}$,

$$\mathrm{f}^C(\vec{q})\ \mathcal{R}_{\mathtt{T}}\ \mathrm{f}^A(\vec{r}), \tag{2.5}$$

- for all program operation symbols $\mathtt{g} \in POPS$, such that $ResSort(\mathtt{g}, \vec{S}) = \mathtt{T}$,

$$\forall (q' \in \mathtt{g}^C(\vec{q}))\exists (r' \in \mathtt{g}^A(\vec{r}))\ q'\ \mathcal{R}_{\mathtt{T}}\ r'. \tag{2.6}$$

Coercion: for all sorts `S` and `T`,

$$(\mathtt{S} \leq \mathtt{T}) \Rightarrow (\forall (q \in \mathtt{S}^C)\exists (r \in \mathtt{T}^A)\ q\ \mathcal{R}_{\mathtt{T}}\ r) \tag{2.7}$$

$$(\mathtt{S} \leq \mathtt{T}) \Rightarrow (\mathcal{R}_{\mathtt{S}} \subseteq \mathcal{R}_{\mathtt{T}}). \tag{2.8}$$

Bistrict: for each sort `T`, $\bot\,\mathcal{R}_{\mathtt{T}}\,\bot$ and whenever $q\,\mathcal{R}_{\mathtt{T}}\,r$ and one of q or r is $\bot$, then so is the other.

V-identical: for each $\mathtt{T} \in SORTS$, if $q\,\mathcal{R}_{\mathtt{T}}\,r$ and either q or r has a visible type, then $q = r$; for each $v \in V$, $\mathcal{R}_v$ contains the identity relation on the carrier set of v (which is the same in both C and A).

The signature Σ is sometimes omitted if it is clear from context.

As noted above, the most important property is the substitution property, which ensures that the simulation is preserved by the specification functions and the program operations. Note that in the definition above, the tuples $\vec{q}$ and $\vec{r}$ may be empty. Therefore, if $\mathcal{R}$ is a simulation relation between C and A, then for all nullary program operation symbols (i.e., type symbols) $\mathtt{g} \in POPS$ such that $ResSort(\mathtt{g}, \langle\rangle) = \mathtt{TClass}$,

$$\mathtt{g}^C()\ \mathcal{R}_{\mathtt{TClass}}\ \mathtt{g}^A() \tag{2.9}$$

The same holds for nullary specification function symbols.

The "coercion" properties are technical requirements. It is necessary for the soundness of verification to be able to relate at each type `T`, each element of every subtype of `T` to some element of type `T`. Furthermore, when one "coerces" two elements at once, any relationships must be preserved. This last property also embodies the intuition that if one object simulates another at a subtype, then this simulation relationship should hold at each supertype, since no extra operations will be applicable at the supertype.

The "bistrict" condition ensures that the meaning of $\bot$ is preserved. It is part of the definition of simulation relations because nontermination is (in some sense) visible.

The "V-identical" property ensures that a simulation relation is (in a certain sense) the identity on objects of the visible types. The first condition says that distinct elements of the visible types cannot be related. The second condition says that at each visible type v, $\mathcal{R}_v$ is the identity on the carrier set of v.

For notational convenience, the following abbreviations are used when dealing with families of relations, $\mathcal{R}$, and the binary relations $\mathcal{R}_{\mathtt{T}}$.

- The notation $Q\ \mathcal{R}_{\mathtt{T}}\ R$ means that for all $q \in Q$, there is some $r \in R$ such that $q\ \mathcal{R}_{\mathtt{T}}\ r$. This notation is often used when comparing sets of possible results and the carrier sets of the various types. As an example of the former use, the substitution property for program operation symbols (Formula 2.6) can be written as follows:

$$\mathtt{g}^C(\vec{q})\ \mathcal{R}_{\mathtt{T}}\ \mathtt{g}^A(\vec{r}).$$

  As an example of the latter use, $\mathtt{S}^C\ \mathcal{R}_{\mathtt{T}}\ \mathtt{T}^A$ means that for each instance q of type `S` in the algebra C, there is an instance r of type `T` in A such that $q\ \mathcal{R}_{\mathtt{T}}\ r$.

- The notation $\vec{q}\mathcal{R}_{\vec{\mathtt{T}}}\vec{r}$ means that for each i, $q_i\mathcal{R}_{\mathtt{T}_i}r_i$ (assuming that $\vec{\mathtt{T}}$ is a tuple of types and $\vec{q}$ and $\vec{r}$ are tuples of objects the same length). This notation was used implicitly in the definition of simulation relations.

Example 2.2.2. The following is an example of a simulation relation.

Let B be the algebra of Figure 2.3 which models the types `IntSet` and `Interval`. For this algebra, the presumed subtype relation $\leq$ is the smallest reflexive relation on the sorts of the specification IPT such that `Interval` $\leq$ `IntSet`. Let $\mathcal{R}$ be the smallest bistrict sorted family of relations between B and B such that for all types `T`, if $q \in \mathtt{T}^B$, then $q \mathcal{R}_{\mathtt{T}} q$, and for all proper $x \leq y$ in $\mathtt{Int}^A$,

$$[x,y] \quad \mathcal{R}_{\mathtt{IntSet}} \quad \mathrm{toSet}^B([x,y]) \tag{2.10}$$

$$[x,y] \quad \mathcal{R}_{\mathtt{IntSet}} \quad [x,y]. \tag{2.11}$$

Notice that $\mathcal{R}_{\mathtt{IntSet}}$ is not symmetric, because a `IntSet` cannot be related to an `Interval` (because of the `choose` operation is more nondeterministic on `IntSet` arguments). This $\mathcal{R}$ is a simulation relation. By construction, $\mathcal{R}$ is bistrict, V-identical, $\mathcal{R}$ relates each element of type `Interval` to some element of type `IntSet`, and relationships at type `Interval` hold at type `IntSet`. To show that $\mathcal{R}$ satisfies the substitution property, let a program operation symbol `g` and $\vec{\mathtt{S}}$ be given such that $ResSort(\mathtt{g}, \vec{\mathtt{S}}) = \mathtt{T}$. (The case for the specification functions is similar.) Let $\vec{q}, \vec{r} \in \vec{\mathtt{S}}^A$ be such that $\vec{q} \mathcal{R}_{\vec{\mathtt{S}}} \vec{r}$. If none of the types in $\vec{\mathtt{S}}$ are `IntSet`, then by construction $\vec{q} = \vec{r}$ and thus $\mathtt{g}^B(\vec{q}) = \mathtt{g}^B(\vec{r})$; since each $\mathcal{R}_{\mathtt{T}}$ contains the identity relation on $\mathtt{T}^B$, it follows that $\mathtt{g}^B(\vec{q})\ \mathcal{R}_{\mathtt{T}}\ \mathtt{g}^B(\vec{r})$.

For the other case, the only program operation symbols that can take an argument of nominal type `IntSet` are `ins`, `elem`, `choose`, `size`, and `remove`. Only `ins` and `choose` are considered below, the rest are similar. Note that if $q_1 \in \mathtt{Interval}^B$, then $q_1\ \mathcal{R}_{\mathtt{IntSet}}\ \mathtt{toSet}^B(q_1)$, by construction. Furthermore, if $x \leq i \leq y$, then inserting into an `Interval` with the program operation `ins` and coercing with "toSet" has the same effect as coercing first and then inserting, which can be seen as follows.

$$\begin{aligned}
&\mathrm{insert}^B([x,y], i) \\
&\quad = \mathrm{insert}^B(\mathrm{toSet}^B([x,y]), i) && (2.12)\\
&\quad = \mathrm{toSet}^B([x,y]) \cup \{i\} && (2.13)\\
&\quad = \mathrm{toSet}^B([x,y]) && (2.14)\\
&\quad = \mathrm{toSet}^B(\mathtt{ins}^B([x,y], i)) && (2.15)\\
&\mathrm{insert}^B([x,y], y+1) \\
&\quad = \mathrm{insert}^B(\mathrm{toSet}^B([x,y]), y+1) && (2.16)\\
&\quad = \mathrm{toSet}^B([x,y]) \cup \{y+1\} && (2.17)\\
&\quad = \mathrm{toSet}^B([x,y+1]) && (2.18)\\
&\quad = \mathrm{toSet}^B(\mathtt{ins}^B([x,y], y+1)) && (2.19)\\
&\mathrm{insert}^B([x,y], x-1) \\
&\quad = \mathrm{insert}^B(\mathrm{toSet}^B([x,y]), x-1) && (2.20)\\
&\quad = \mathrm{toSet}^B([x,y]) \cup \{x-1\} && (2.21)\\
&\quad = \mathrm{toSet}^B([x-1,y]) && (2.22)\\
&\quad = \mathrm{toSet}^B(\mathtt{ins}^B([x,y], x-1)) && (2.23)
\end{aligned}$$

So if $q_1\ \mathcal{R}_{\mathtt{IntSet}}\ r_1$ and $q_2\ \mathcal{R}_{\mathtt{Int}}\ r_2$, then

$$\mathtt{ins}^B(q_1, q_2) \quad \mathcal{R}_{\mathtt{IntSet}} \quad \mathtt{ins}^B(r_1, r_2) \tag{2.24}$$

because if either q_1 or r_1 is an `Interval`, it can be replaced by the `IntSet` it simulates, and the result will be simulated by the original result. Since $\mathcal{R}_{\mathtt{IntSet}}$ is transitive, the result follows. For `choose`, one must first observe that if q_1 is an `Interval`, then

$$q = \mathtt{leastElement}^B(q_1) \Rightarrow q \in \mathtt{toSet}^B(q_1). \tag{2.25}$$

Suppose $q_1\ \mathcal{R}_{\mathtt{IntSet}}\ r_1$, then for all $q \in \mathtt{choose}^B(q_1)$, there is some $r \in \mathtt{choose}^B(r_1)$ such that $q\ \mathcal{R}_{\mathtt{Int}}\ r$. That is, if q_1 is of type `Interval`, then q is the least element of q_1, and thus must be an element of r_1 (furthermore, it is a possible result on r_1, regardless of the type of r_1). On the other hand, if q_1 is of type `IntSet`, then so is r_1, since $q_1\ \mathcal{R}_{\mathtt{IntSet}}\ r_1$; so $q_1 = r_1$.

So this $\mathcal{R}$ is a simulation relation.

Example 2.2.3. It is not always possible to find a simulation between an algebra and itself. Consider an algebra C that is just like B of Figure 2.3, except that the `choose` operation is deterministic and returns the maximum element of a non-empty `IntSet`. Consider the relation $\mathcal{R}$ defined in example 2.2.2, with C substituted for B. This $\mathcal{R}$ is not a simulation relation between C and C, because $[1,3]\,\mathcal{R}_{\mathtt{IntSet}}\,\{1,2,3\}$, but

$$\{1\} = \mathtt{choose}^C([1,3]) \tag{2.26}$$

$$\{3\} = \mathtt{choose}^C(\{1,2,3\}) \tag{2.27}$$

and 1 is not related by $\mathcal{R}_{\mathtt{Int}}$ to 3.

The substitution property for program operations is the key to the inductive proof that simulation is preserved by all programming language expressions. The substitution property for specification functions is the key to the proof that simulation is preserved by all assertions and the soundness of the modular program verification techniques described in Chapter 6.

A substitution property is also used to define homomorphisms between multisorted algebras [EM85]. The substitution property defined above differs from the usual substitution property for multisorted algebras in two ways: it is a substitution property for relations instead of functions, and it allows relations among objects of different types.

Simulation relations resemble the "logical relations" used in the study of the lambda calculus [Sta85] [Mit86]. An important property of logical relations is captured by the so-called fundamental theorem of logical relations [Sta85], which states that simulation is preserved by expressions.

The substitution property is similar to the defining property of Nipkow's simulation relations [Nip86]. The difference is that we provide for message passing (through program operations) and allow objects of one type to be related to objects of another type.

Simulation relations are also similar to, but more general than, the coercer functions of Reynolds [Rey80] [Rey85] and Bruce and Wegner [BW87a]. These authors also require a substitution property.

Chapter 3

Polymorphic Type and Function Specifications

In this chapter a new method for the modular specification of abstract types and polymorphic functions that use message passing is described. The specification language is adapted from Wing's interface specification language for CLU [Win83] [LG86, Chapter 10] [GHW85] [Win87] and Chen's Larch/Generic interface specification language [Che89]. However, unlike Wing, the specifications only deal with immutable types.

The specification of a function or a program operation is written as if each argument and result has the specified type. However, actual arguments and results are allowed to have types that are subtypes of the specified types. This follows the practice of Trellis/Owl [SCB+86] and other typed object-oriented programming languages. An example is the specification of *inBoth*, found in Figure 1.6. It specifies that the arguments may be instances of a subtype of `IntSet`, but its post-condition is written using the specification functions that describe the abstract values of the type `IntSet`. The advantage of this approach is that the syntax and semantics of specifications parallels that of the implementations. The problem is to give meaning to such specifications when actual arguments do not have the specified types (for example, when an argument has the type `Interval`). The solution is to require that the subtypes all be specified so that specifications tailored to their supertypes are meaningful. Technically this is done by requiring that abstract models of a subtype can interpret assertions written for their supertypes, as formalized in the restrictions on signatures given in Chapter 2.

The meaning of a specification is independent of the definition of subtype relations (given in Chapter 4). This independence is appropriate, since the semantic restrictions that make a preorder on types a subtype relation are intended to ensure the soundness of particular program verification techniques.

The syntax and semantics of type specifications are presented first, followed by the syntax and semantics of polymorphic function specifications. A discussion follows these sections.

3.1 Type Specifications

A set of abstract type specifications has two purposes:

- describing a family of algebras, and
- describing how the objects of a subtype simulate the objects of their supertypes.

The type specification syntax is specialized for class-based object-oriented languages such as Smalltalk-80 or Trellis/Owl. Hence each type specification really describes two types: one for the class object and one for the instances of that class. These types are, of course, closely connected. The syntax suppresses the implicit operation that returns the class object; instead one specifies class and instance operations. Class operations take a class object as an argument. Instance operations take an instance of the specified type as an argument. Both are program operations.

Abstractly, a set of type specifications has five parts: a set of sort symbols, a binary relation on these sort symbols, a specification of a family of relations among abstract values, a set of traits, and a specification of the program operations of a model. The binary relation on sorts is called a *presumed subtype relation*, since it should relate subtypes to supertypes. The family of relations among abstract values is intended to be a simulation relation; this family of relations will be used to show that the presumed subtype relation is a subtype relation. Each trait describes the abstract values of a type; formally it specifies the carrier set and specification functions of a model. (Unless these traits are unconventional, they will be found in the Larch Shared Language Handbook [GH86a].) The bulk of the specifications is taken up by program operation specifications.

In what follows sets of type specifications will often be named. For example, the set of type specifications named II consists of the specifications `IntSet` found in Figure 1.1 and `Interval` found in Figure 1.2. The specification II is used below to help explain the specification language. This specification might also be called `IntSet + Interval`, and this notation will be used on occasion.

The syntax of sets of type specifications is described first then how a set of type specifications determines the signature of the algebras in its semantics, and their semantics.

3.1.1 Type Specification Syntax

In Figure 3.1 the syntax of sets of type specifications is presented. The nonterminal symbols ⟨specification function⟩, ⟨type⟩, and ⟨identifier⟩ represent specification function symbols, type symbols, and program operation symbols and other identifiers (respectively). The nonterminal ⟨term⟩ is as in [GH86b]; in addition, terms of the form $e_1 = e_2$ are allowed. However, in specifications, equality (=) may only be used between terms of visible sort; such terms are called *subtype-constraining*. Specification function symbols that are declared to be infix may be written between terms, specification function symbols that are

declared to be postfix may be written after terms, and so on. The mixfix symbol **if then else fi** has the lowest precedence, all others need to be parenthesized to prevent ambiguity.

A set of type specifications is formally a list of ⟨type spec⟩s, each of which specifies an abstract type. A ⟨type spec⟩ has a type name, a list of subtype information, a list of the program operation symbols that name the type's class and instance operations, a ⟨basis⟩ clause, and a list of program operation specifications. The ⟨basis⟩ clause tells what trait is used in the program operation specifications.

The behavior of a program operation is described using a ⟨pre-condition⟩ and an ⟨post-condition⟩. These clauses contain boolean ⟨term⟩s written using specification function symbols, the formal arguments of the operation, and equations. The post-condition may also contain the formal result identifier, although this identifier cannot be used in the pre-condition. These terms must sort-check. Sort checking for terms is defined inductively. Each identifier appearing in a term has a nominal sort, given in its declaration (as a formal argument or formal result). The nominal sort of a specification function application of the form $f(\vec{e})$ is a sort `T` if the nominal sort of $\vec{e}$ is $\vec{S}$, $ResSort(f, \vec{S})$ is defined, and $ResSort(f, \vec{S}) = T$, otherwise the term does not sort-check. (How the *ResSort* map is determined from the traits is described below.) The nominal sort of an equation $e_1 = e_2$ is `Bool` if e_1 and e_2 have the same nominal sort, otherwise the equation does not sort-check. The nominal sort of the term in a pre- or post-condition must be `Bool`.

A term that sort-checks and only uses specification functions of a signature Σ is a Σ-*term*. Of particular interest are terms of nominal sort `Bool`.

Definition 3.1.1 (Σ-assertion). A Σ-*assertion* is a Σ-term of nominal sort `Bool`.

For convenience, the following syntactic sugars are defined. An ⟨specification function⟩ such as "f" used in a ⟨term⟩ without arguments is syntactic sugar for "f()." A declaration such as "f,s: Int" is syntactic sugar for the declaration list "f: Int, s: Int." Furthermore, an omitted ⟨pre-condition⟩ is syntactic sugar for a pre-condition of the form "**requires** true." A syntactic sugar for exceptions is discussed in the last section of this chapter.

3.1.2 Signature of a Type Specification

The semantics of a set of type specifications is a family of algebras with the same signature. If *SPEC* is a set of type specifications, then *SIG*(*SPEC*) is the signature determined by *SPEC*.

A program operation symbol may be present in many different type specifications. This convention allows programmers to exploit message passing. For example, in the specification II, `ins` is an instance operation of both `IntSet` and `Interval`. Each of occurrence of a program operation symbol in a set of type specifications is associated with a different *nominal signature*, which is a pair consisting of a tuple of type symbols and a type symbol, written $S_1, \ldots, S_n \rightarrow T$ or $\vec{S} \rightarrow T$ (or $\rightarrow T$ if there are no arguments).

⟨set of type specifications⟩ ::= ⟨type spec list⟩
⟨type spec list⟩ ::= ⟨type spec⟩
    | ⟨type spec list⟩ ⟨type spec⟩

⟨type spec⟩ ::= ⟨type⟩ **immutable type**
        ⟨subtype list⟩
        ⟨class ops⟩ ⟨instance ops⟩
        ⟨basis⟩
        ⟨prog op spec list⟩

⟨subtype list⟩ ::= ⟨empty⟩
    | ⟨subtype clause⟩ ⟨subtype list⟩
⟨empty⟩ ::=
⟨subtype clause⟩ ::= **subtype of** ⟨type⟩
        by ⟨term⟩ **simulates** ⟨term⟩

⟨class ops⟩ ::= ⟨empty⟩
    | **class ops** [⟨ident list⟩]
⟨instance ops⟩ ::= **instance ops** [⟨ident list⟩]

⟨ident list⟩ ::= ⟨identifier⟩
    | ⟨ident list⟩ , ⟨identifier⟩

⟨basis⟩ ::= **based on sort** ⟨identifier⟩
        from ⟨identifier⟩ ⟨with clause⟩
⟨with clause⟩ ::= ⟨empty⟩
    | **with** [⟨renaming list⟩]
⟨renaming list⟩ ::= ⟨renaming⟩
    | ⟨renaming list⟩ , ⟨renaming⟩
⟨renaming⟩ ::= ⟨identifier⟩ **for** ⟨ident list⟩

⟨prog op spec list⟩ ::= ⟨prog op spec⟩
    | ⟨prog op spec list⟩ ⟨prog op spec⟩
⟨prog op spec⟩ ::= **op** ⟨nominal signature⟩
        ⟨pre-condition⟩ ⟨post-condition⟩
⟨nominal signature⟩ ::= ⟨identifier⟩ (⟨decl list⟩)
        returns (⟨decl⟩)
⟨decl list⟩ ::= ⟨decl⟩ | ⟨decl list⟩ , ⟨decl⟩
⟨decl⟩ ::= ⟨identifier⟩ : ⟨type⟩

⟨pre-condition⟩ ::= **requires** ⟨term⟩
⟨post-condition⟩ ::= **ensures** ⟨term⟩

⟨term list⟩ ::= ⟨term⟩ | ⟨term list⟩ , ⟨term⟩

Figure 3.1: Syntax of Type Specifications.

The parts of a set of type specification's signature are determined as follows. (The reader may wish to consult the signature of II given in Figure 2.1 during this discussion.)

The set of visible types, V, is fixed by convention. For concreteness, the visible types determined by a set of type specification will always be as follows:

$$V \stackrel{\text{def}}{=} \{\texttt{Bool}, \texttt{Int}, \texttt{IntStream}, \texttt{BoolStream}\}. \quad (3.1)$$

The set of type symbols, *TYPES*, described by a type specification consists of the visible types, the type symbols named at the beginning of each ⟨type spec⟩, and a *class type* for each of the types already mentioned, formed by adding "`Class`" as a suffix to each of the other type symbols. For example, the set of type symbols of II includes `IntSet`, `Interval`, `IntSetClass`, and `IntervalClass`, in addition to the visible types and their associated class types: `BoolClass`, `IntClass`, `IntStreamClass`, and `BoolStreamClass`.

The set of sorts *SORTS* and the set of specification function symbols, *SFUNS* are determined by the traits referenced in the ⟨basis⟩ clauses of a specification, plus a trait of the following form for each class type `TClass`:

TClass: **trait**
  introduces T: → TClass
    #==#: TClass, TClass → Bool
  asserts for all [t_1,t_2: TClass]
    $t_1 = t_2$
    $t_1 == t_2$

The traits referenced in the specification of `IntSet` and `Interval` are found in Figures 3.2 and 3.3. These traits introduce various sorts and operations, which become the sorts and specification function symbols, as renamed by the ⟨renaming⟩ clause. The imported traits are from [GH86a]. However, the sort name following the keyword **sort** in the specification of a type named `T` is renamed to `T`. In the II example, there is an auxiliary sort `Card` that has no corresponding type. It is used in the specification of the program operation `size`. So in general there will be more sorts than types.

The presumed subtype relation, $\leq$, is the reflexive, transitive closure of the relationships mentioned in the ⟨subtype clause⟩s of each type. For example II states that `Interval` is a subtype of `IntSet`. Hence `Interval` $\leq$ `IntSet`. By taking the reflexive, transitive closure, the relationship `IntSet` $\leq$ `IntSet` holds, as does `Bool` $\leq$ `Bool`, and so on.

The requirement on signatures that the *ResSort* mapping is monotone in $\leq$ does not affect the construction of *ResSort*. However, if this requirement is not met, then the set of type specifications is invalid, as it will not determine a proper signature. Thus it is left to the designer to specify the specification functions and program operations of presumed subtypes so that signature restrictions are met. (Some automation of this task would help specifiers.)

The result sort mapping *ResSort* for specification function symbols is determined as follows. Each specification function symbol is introduced in a trait along with a signature (e.g., $\vec{\texttt{S}} \rightarrow \texttt{T}$). The mapping *ResSort* is simply another representation for this information. So, if the specification function symbol f is introduced

IntSetTrait: **trait**
  imports SetBasics **with** [Int **for** E],
    SetIntersection **with** [Int **for** E],
    isEmpty **with** [Int **for** E, {} **for** new],
    Singleton **with** [Int **for** E, {} **for** new,
      {#} for singleton],
    Join **with** [Int **for** E, {} **for** new,
      ∪ for .join],
    CardToInt
  introduces #==#: C,C → Bool
  asserts for all [s_1, s_2: C]
    $(s_1 == s_2) = (s_1 = s_2)$

CardToInt: **trait**
  imports Cardinal, Integer
  introduces toInt: Card → Int
  asserts for all [c: Card]
    toInt(0) = 0
    toInt(succ(c)) = 1 + toInt(c)

Figure 3.2: The traits `IntSetTrait` and `CardToInt`.

with signature $\vec{\texttt{S}} \rightarrow \texttt{T}$, then *ResSort* maps the symbol f and tuple of sorts $\vec{\texttt{S}}$ to `T`.

The set of program operation symbols *POPS* of a set of type specifications consists of the symbols following **op** in ⟨prog op spec⟩s, all type symbols that are not class types, and program operation symbols for the visible types. For example, the set of program operation symbols of II includes `null`, `create`, `ins`, `elem`, `IntSet`, `Interval`, `Bool`, `Int`, `IntStream`, `BoolStream`, and program operation symbols for the visible types such as `true`, `false`, `not`, `and`, `or`, `add`, and so on. A type symbol such as `IntSet` is a nullary program operation symbol, which is used to access a class object.

The program operation symbols associated with the visible types are found in Figures B.1, B.2, and B.3 (with a similar set of operations for `BoolStream`). For example, in Figure B.1, the program operation `or` is defined, which means that a program operation symbol `or` is associated with `Bool`.

Each instance operation in a given type's specification must have at least one argument with the specified type. Similarly each class operation must have an argument of the corresponding class type. By convention, these required arguments are listed as the first argument, which makes it easier for humans to infer what operation will be invoked. This convention is enforced in Trellis/Owl and Smalltalk-80, but it not enforced in CLOS or by the specification language.

Each ⟨prog op spec⟩ declares a *nominal signature*. This nominal signature is formed by placing an arrow between the list of the types in the arguments part of the operation signature and the type in the return part. For example, the nominal signature of the program operation `ins` in the specification of `IntSet` is

`IntSet`, `Int` → `IntSet`.

IntervalTrait: **trait**
imports IntSetTrait **with** [IntSet **for** C]
introduces [#,#]: Int, Int → C
insert, delete: C, Int → IntSet
size: C → Card
#∈#: C,Int → Bool
isEmpty: C → Bool
∪, ∩: C,C → IntSet
∪, ∩: C,IntSet → IntSet
∪, ∩: IntSet,C → IntSet
#==#: C,C → Bool
#==#: C,IntSet → Bool
#==#: IntSet,C → Bool
toSet: C → IntSet
leastElement, greatestElement: C → Int
asserts for all [c, c_1: C, s: IntSet, x, y, i: Int]
$[x,y]$ = **if** $x \leq y$ **then** $[x,y]$ **else** $[x,x]$ **fi**
insert($[x,y]$, i) = insert(toSet($[x,y]$),i)
delete($[x,y]$, i) = delete(toSet($[x,y]$),i)
size($[x,y]$) = size(toSet($[x,y]$))
$(i \in [x,y]) = (i \in \text{toSet}([x,y]))$
isEmpty($[x,y]$) = false
leastElement($[x,y]$) = x
$(s == c) = (s == \text{toSet}(c))$
$(c == s) = (s == \text{toSet}(c))$
$(c == c_1) = (\text{toSet}(c) == \text{toSet}(c_1))$
$(s \cap c) = (s \cap \text{toSet}(c))$
$(c \cap s) = (s \cap \text{toSet}(c))$
$(c \cap c_1) = (\text{toSet}(c) \cap \text{toSet}(c_1))$
$(s \cup c) = (s \cup \text{toSet}(c))$
$(c \cup s) = (s \cup \text{toSet}(c))$
$(c \cup c_1) = (\text{toSet}(c) \cup \text{toSet}(c_1))$
greatestElement($[x,y]$) = **if** $x \leq y$ **then** y
else x **fi**
toSet($[x,y]$) = **if** $y \leq x$ **then** $\{x\}$
else insert(toSet($[x,y-1]$), y) **fi**

Figure 3.3: The trait `IntervalTrait`.

Class operations also have a nominal signature, for example, the nominal signature of the `create` operation in the specification of `Interval` is

$$\texttt{IntervalClass},\texttt{Int},\texttt{Int} \rightarrow \texttt{Interval}.$$

Finally, type symbols, such as `IntSet`, are nullary program operations. Hence their nominal signature is such that there are no arguments and the result type is the corresponding class type. For example, the nominal signature of the `IntSet` operation is:

$$\rightarrow \texttt{IntSetClass}.$$

The result sort map, *ResSort*, for program operation symbols is determined from the presumed subtype relation $\leq$ and the nominal signature of each program operation specification as follows. For each program operation symbol $\mathtt{g}$, and each tuple of sorts $\vec{\mathtt{S}}$, $\mathit{ResSort}(\mathtt{g}, \vec{\mathtt{S}})$ is defined and equal to some sort $\mathtt{T}$ if and only if there is a unique signature $\vec{\mathtt{U}} \rightarrow \mathtt{T}$ that is the nominal signature of a program operation specification whose operation symbol is $\mathtt{g}$ such that $\vec{\mathtt{S}} \leq \vec{\mathtt{U}}$ and for all operation specifications whose operation symbol is $\mathtt{g}$, if the tuple of nominal argument types $\vec{\mathtt{V}}$ has the same length as $\vec{\mathtt{S}}$ and if $\vec{\mathtt{S}} \leq \vec{\mathtt{V}}$, then $\vec{\mathtt{U}} \leq \vec{\mathtt{V}}$. (The formula $\vec{\mathtt{U}} \leq \vec{\mathtt{V}}$ means that for each i, $\mathtt{U}_i \leq \mathtt{V}_i$.) Hence the unique program operation specification with the most specific argument type requirements that apply to $\vec{\mathtt{S}}$ determines the nominal result type. Furthermore, there must be a single such program operation specification for each combination of argument types, or *ResSort* is not defined on that combination of argument types.

The use of *ResSort* for determining the result sort of a program operation allows the specification of binary operations where the code executed depends on the types of more than one argument. Consider a specification of types `Rat` and `Int`, where `Int` $\leq$ `Rat`. The operation `add` may be specified for the type `Rat` with the nominal signature

$$\texttt{Rat}, \texttt{Rat} \rightarrow \texttt{Rat}$$

while the operation `add` may be specified for the type `Int` with the nominal signature

$$\texttt{Int}, \texttt{Int} \rightarrow \texttt{Int}.$$

Then *ResSort* behaves as follows:

$$\begin{aligned}
\mathit{ResSort}(\texttt{add}, \langle \texttt{Rat}, \texttt{Rat} \rangle) &= \texttt{Rat} \\
\mathit{ResSort}(\texttt{add}, \langle \texttt{Int}, \texttt{Rat} \rangle) &= \texttt{Rat} \\
\mathit{ResSort}(\texttt{add}, \langle \texttt{Rat}, \texttt{Int} \rangle) &= \texttt{Rat} \\
\mathit{ResSort}(\texttt{add}, \langle \texttt{Int}, \texttt{Int} \rangle) &= \texttt{Int}
\end{aligned}$$

In Trellis/Owl, one would specify `add` for the type `Rat` as above, but for the type `Int` the operation `add` would be specified with the nominal signature:

$$\texttt{Int}, \texttt{Rat} \rightarrow \texttt{Rat}.$$

For such a specification,

$$\mathit{ResSort}(\texttt{add}, \langle \texttt{Int}, \texttt{Int} \rangle) = \texttt{Rat}.$$

which is less specific than might be desired. Various researchers have used "bounded quantification" to state type restrictions so that the nominal type of adding two integers is an integer [CW85] [Car88], but bounded quantification has its problems [BL88].

On the other hand, the designer must be careful to specify types in such a way that a legal signature results. For example, if the operation `add` were specified for the type `Rat` with the nominal signature

$$\texttt{Rat}, \texttt{Int} \rightarrow \texttt{Rat}$$

and the operation `add` were specified for the type `Int` with the nominal signature

$$\texttt{Int}, \texttt{Rat} \rightarrow \texttt{Int},$$

then $ResSort(\texttt{add}, \langle \texttt{Int}, \texttt{Int} \rangle)$ would be undefined.

3.1.3 Satisfaction for Type Specifications

The definition of when an algebra satisfies a set of type specifications involves showing that the algebra has the right signature, that its trait structure satisfies the specification's traits, that its model of the visible types is standard, and that for arguments having appropriate types each program operation satisfies its specification. These definitions rely on the monotonicity (in the presumed subtype relation, $\leq$) of the result sort mapping. That is, since the assertions in a specification are meaningful for arguments and results of the specified types, and since the specification functions involved must be applicable to subtypes, the specification functions can be applied to all subtypes as well.

The first part of satisfaction is showing that the trait structure of an algebra satisfies the traits of the specification. As indicated in Chapter 2, the trait structure is formed by deleting $\perp$ from each sort's carrier set, restricting the specification functions to these domains, and throwing out the program operations. The specification functions are defined on carrier sets without $\perp$ because they are strict and their inverses are strict. A trait structure is thus a multi-sorted algebra, as used in the semantics of first-order logic [End72].

An algebra A *satisfies the traits* of a specification if and only if the trait structure of A is a model of those traits in the usual sense of models of first-order formulas in formal logic.

The visible types have a standard model which is the algebra, B, that combines Figures B.1 (`Bool`), B.2 (`Int`), and B.3 (`IntStream`) and a model of `BoolStream` that is analogous to `IntStream`. These are discussed in Appendix B. Let $SIG(B)$ be the signature of B.

An algebra has the standard model of the visible types if its $SIG(B)$-reduct is the algebra B described above. (The signature determined by a specification automatically has $SIG(B)$ as a subsignature.)

The definition of satisfaction for program operation specifications relies on the details of evaluation of assertions. Evaluation of assertions is based on environments.

Environments are mappings that give meaning to the free identifiers in an assertion. The only identifiers that can appear in an assertion are the formal arguments and formal result identifiers. Each such identifier has a nominal type, given by its declaration (as a formal argument or result). Hence an environment's domain consists of identifiers of particular types.

Definition 3.1.2 (Σ-environment). Let Σ be a signature, whose set of type symbols is $TYPES$, and whose presumed subtype relation is $\leq$. Let A be a Σ-algebra. Let X be a set of identifiers indexed by $TYPES$. Then a mapping $\eta: X \rightarrow |A|$ is a Σ-*environment* if and only if for every type $\texttt{T} \in TYPES$ and for every $\texttt{x}$ of nominal type $\texttt{T}$ in X, $\eta(\texttt{x})$ has a type $\texttt{S}$ such that $\texttt{S} \leq \texttt{T}$.

The signature (Σ) is omitted when it is clear from context.

An environment may map an identifier `x: T` to an object of the carrier set of an algebra, only if the value of `x` has some type `S` such that is a presumed subtype of `T`. Emphasis is laid on this condition by saying that an environment *obeys* a presumed subtype relation.

Allowing environments to map identifiers of one type to objects of another type is the major technical difference between the above definition and the definition of environments used in traditional semantics. The traditionally restricted environments are called "nominal", since the nominal type of an identifier in their range determines the type of that identifier's value.

Definition 3.1.3 (nominal Σ-environment). Let Σ be a signature. A *nominal* Σ-*environment*, η, is a Σ-environment such that for each `x:T`, $\eta(\texttt{x})$ has type `T`.

An environment is *proper* if its range does not include $\perp$. In standard semantics, a proper and nominal environment is often called an assignment, when used to give meaning to terms.

The following shorthand is used for adding a binding to an environment:

$$\eta[q/\texttt{x}] \stackrel{\text{def}}{=} \lambda l.\ \text{if } l = \texttt{x} \text{ then } q \text{ else } \eta(l). \tag{3.2}$$

(Of course, this shorthand only makes sense if q has some type `S` that is a presumed subtype of the nominal type of `x`.)

The basis for evaluation of assertions is the following definition of the extension of an environment to an evaluation of terms whose free identifiers are in the domain of the environment. This extension uses the specification functions of the algebra in the environment's range to evaluate specification function symbols and uses the environment itself to evaluate free identifiers [EM85, Section 1.10].

Let Σ be a signature and A a Σ-algebra. The notation $\overline{\eta}$ means the extension of the Σ-environment $\eta : X \rightarrow |A|$ to a mapping from terms that sort-check to elements of the carrier set of A. For example, if `p` has nominal sort `IntSet`, `i` : `Int`, $\eta(\texttt{s}) = \{1, 2, 3\}$, $\eta(\texttt{i}) = 1$, and $\# \in \#^A(\{e_1, \ldots, e_n\}, e_i) = true$, then

$$\begin{aligned} \overline{\eta}[\![\texttt{i} \in \texttt{s}]\!] &= \# \in \#^A(\eta(\texttt{i}), \eta(\texttt{s})) \\ &= \# \in \#^A(1, \{1, 2, 3\}) \\ &= true. \end{aligned}$$

Furthermore, let

$$\overline{\eta}[\![E_1 = E_2]\!] \stackrel{\text{def}}{=} \begin{cases} true & \text{if } \overline{\eta}[\![E_1]\!] = \overline{\eta}[\![E_2]\!] \\ false & \text{otherwise.} \end{cases} \tag{3.3}$$

It is assumed that the carrier set of `Bool` is as in Figure B.1 in all algebras; hence *true* and *false* are objects in all algebras.

The extension $\overline{\eta}$ of an environment η is well-defined, because of the requirements on signatures given in Chapter 2. One might worry that the above definition is nonsense, since an environment need not be nominal, and hence a specification function might be applied outside its domain. That cannot happen, however, since the definition of signatures ensures that if an assertion sort-checks, then the specification functions involved will only be applied within their domains. The key to showing this formally is the following lemma, whose proof is by induction on the structure of terms.

Lemma 3.1.4. Let Σ be a signature, whose presumed subtype relation is $\leq$. Let P be an term written using the specification functions of Σ and whose set of free identifiers is X. Let C be a Σ-algebra. Let $\eta: X \rightarrow |C|$ be a Σ-environment.

If P has nominal sort $\mathtt{T}$, then $\overline{\eta}[\![P]\!]$ has some sort $\mathtt{S} \leq \mathtt{T}$. ∎

The above lemma can be used to show that no specification function is applied outside its domain when evaluating an assertion that sort-checks as follows. Consider an application $\mathrm{f}(\vec{e})$ in an assertion. Since this application sort-checks, the arguments $\vec{e}$ must have some nominal sort $\vec{\mathtt{S}}$, such that $\mathit{ResSort}(\mathrm{f}, \vec{\mathtt{S}}) = \mathtt{T}$ for some sort $\mathtt{T}$. By the above lemma, $\vec{e}$ must have some sort $\vec{\mathtt{U}} \leq \vec{\mathtt{S}}$. But since *ResSort* is monotonic, $\mathit{ResSort}(\mathrm{f}, \vec{\mathtt{U}})$ is defined, and thus "f" will not be applied outside its domain. The signature restrictions therefore compensate for the freedom allowed in environments.

An algebra-environment pair models an assertion when one can think of the assertion as true in that environment.

Definition 3.1.5 (models). Let Σ be a signature. Let P be an assertion whose set of free identifiers is X. Let C be a Σ-algebra. Let Y be a set of typed identifiers such that $X \subseteq Y$. Let $\eta_C: Y \rightarrow |C|$ be a Σ-environment. Then (C, η_C) *models* P, written $(C, \eta_C) \models P$, if and only if $\overline{\eta_C}[\![P]\!] = \mathit{true}$.

For example, consider the assertion "$1 \in \mathtt{s}$." Let C be the algebra of Figure 2.3. Let Y be the set $\{\mathtt{s} : \mathtt{IntSet}\}$. Let $\eta_C: Y \rightarrow |C|$ be the environment such that $\eta(\mathtt{s}) = \{1, 2, 3\}$. Since $\overline{\eta_C}[\![1 \in \mathtt{s}]\!] = \mathit{true}$ by the above definition, $(C, \eta) \models 1 \in \mathtt{s}$. Since $\overline{\eta_C}[\![4 \in \mathtt{s}]\!] = \mathit{false}$, (C, η_C) does not model "$4 \in \mathtt{s}$."

The above definition of "models" specializes to the standard definition [End72] when the presumed subtype relation is equality ($=$). This follows trivially, since the only difference between the above definition and the standard definition is the notion of extended application.

In the determination of *ResSort* for program operations, the nominal sort of a program operation applied to certain types of arguments was determined by the unique program operation specification with nominal argument types that were the least in the $\leq$ ordering and supertypes of the given argument types. This unique program operation specification is called the *most specific applicable* program operation specification for the given argument types. The behavior of a program operation of an algebraic model for given argument types is determined by the most specific applicable program operation specification. The program operation must satisfy this specification in the sense that if the arguments have types that are presumed subtypes of the nominal argument types and model the pre-condition, then the operation must halt, and can only return results that model the post-condition and that have a type that is a presumed subtype of the nominal result type.

Definition 3.1.6 (satisfies for operations). Let *SPEC* be a set of type specifications. Let $\mathtt{g}$ be a program operation symbol of $\mathit{SIG}(\mathit{SPEC})$. Let C be an algebra whose signature is $\mathit{SIG}(\mathit{SPEC})$. An operation $\mathtt{g}^C$ *satisfies* the specification of an operation $\mathtt{g}$ in *SPEC* if and only if for all tuples of types $\vec{\mathtt{S}}$, if $\mathit{ResSort}(\mathtt{g}, \vec{\mathtt{S}}) = \mathtt{T}$, then the following condition is met. Let O be the most specific applicable program operation specification for $\vec{\mathtt{S}}$. This O has the following form, where $\vec{\mathtt{S}} \leq \vec{\mathtt{U}}$:

op $\mathrm{g}(\vec{\mathtt{x}} : \vec{\mathtt{U}})$ **returns**$(\mathtt{y} : \mathtt{T})$
requires R
ensures Q.

Let $X = \{\mathtt{x}_1 : \mathtt{U}_1, \ldots, \mathtt{x}_n : \mathtt{U}_n\}$. For all proper $\vec{q} \in \vec{\mathtt{S}}^C$, for all $\mathit{SIG}(\mathit{SPEC})$-environments $\eta: X \rightarrow |C|$ such that $\eta(\mathtt{x}_i) = q_i$, if

$$(C, \eta) \models R, \tag{3.4}$$

then for all possible results $r \in \mathtt{g}^C(\vec{q})$:

$$r \neq \bot \tag{3.5}$$

$$(C, \eta[r/\mathtt{y}]) \models Q, \tag{3.6}$$

and there is some $\mathtt{V} \in \mathit{TYPES}$ such that $r \in \mathtt{V}^C$ and $\mathtt{V} \leq \mathtt{T}$. Furthermore, whenever some argument to the operation is $\bot$, then the only possible result is $\bot$.

A tuple of proper arguments *satisfies the pre-condition* "**requires** R" when η is a proper environment that maps the formals to the given tuple of arguments, then $\overline{\eta}[\![R]\!] = \mathit{true}$.

For example, the operation defined by

$$\mathtt{elem}^A(s, i) \stackrel{\text{def}}{=} \{\# \in \#^A(i, s)\} \tag{3.7}$$

$$\mathtt{elem}^A(\bot, i) \stackrel{\text{def}}{=} \{\bot\} \tag{3.8}$$

$$\mathtt{elem}^A(s, \bot) \stackrel{\text{def}}{=} \{\bot\} \tag{3.9}$$

$$\mathtt{elem}^A(\bot, \bot) \stackrel{\text{def}}{=} \{\bot\} \tag{3.10}$$

(where s and i are proper) satisfies the specification of the `elem` operation for the specification II that includes `IntSet` and `Interval` because of the following.

- For all assignments η that bind some proper `IntSet` s to $\mathtt{s}$: `IntSet`, $\overline{\eta}[\![\mathrm{true}]\!] = \mathit{true}$ (recall that an omitted pre-condition is syntactic sugar for "**requires** true") and the only possible result is $\# \in \#^A(i, s)$. This result is proper and satisfies

the post-condition of the most specific applicable program operation specification, which is found in the specification of the type `IntSet`. The satisfaction of the post-condition is shown as follows:

$$\overline{\eta[\# \in \#^A(i,s)/\mathtt{b}]}[\![\mathtt{b} = \mathtt{i} \in \mathtt{s}]\!] = \mathit{true}. \quad (3.11)$$

- Similarly, for all assignments η that bind some proper `Interval` s to `s` : `Interval`, the only possible result is $\# \in \#^A(i,s)$, which is proper and satisfies the post-condition specified for `elem` in the type specification `Interval`.

The nullary operations that name class objects are implicitly specified as follows:

op T() **returns**(y:TClass)
  ensures y == T.

An operation $\mathtt{T}^A$ satisfies this specification if its only possible result is class object for `T`.

An algebra satisfies a set of type specifications if the above conditions are met. This is summarized in the following definition.

Definition 3.1.7 (satisfaction for algebras). Let *SPEC* be a set of type specifications. Let A be a $SIG(SPEC)$-algebra. Then A *satisfies SPEC* if and only if

- the trait structure of A satisfies the traits of *SPEC*,
- the $SIG(B)$-reduct of A is B, where B is the algebra described above that models the visible types, and
- for each of the program operation specifications of *SPEC*, the corresponding program operation of A satisfies its specification.

For example, the algebra of Figure 2.3 satisfies the specification II.

The *semantics of a set of type specifications SPEC* is a set of all $SIG(SPEC)$-algebras, that satisfy *SPEC*. Each algebra in the semantics of *SPEC* is called a *SPEC-algebra*. For example, the algebra of Figure 2.3 is an II-algebra.

3.2 Nondeterministic Type Specifications

Nondeterministic program operations are useful for modeling both "undefined" behavior and types that are inherently nondeterministic. An example is the type `IntSet` specified in Figure 1.1.

As an example of using nondeterminism for modeling "undefined" behavior, consider the `choose` operation of `IntSet`. This program operation has a non-trivial pre-condition. When the pre-condition is not satisfied, the set of possible results is unconstrained, and hence can be entire carrier set of the nominal result type. An example is the II-algebra, B, of Figure 2.3 whose carrier set for the type `IntSet` is the set of finite sets of integers (plus $\perp$), and in which

$$\mathtt{choose}^B(\{\}) \stackrel{\text{def}}{=} \mathtt{Int}^B = \{\perp, 0, 1, -1, \ldots\}. \quad (3.12)$$

That is, when the argument is empty, `choose` may fail to terminate, or may return any integer.

The `choose` operation also illustrates nondeterminism for its own sake, since even when its pre-condition is satisfied, it may return any element of the argument set.

An algebra may satisfy the specification with an implementation that is more deterministic than allowed by the specification. A program operation may be more "defined", as in an algebra A, where

$$\mathtt{choose}^A(\{\}) \stackrel{\text{def}}{=} \{0\}. \quad (3.13)$$

Or the normal case may be made more deterministic, for example returning only the least element of the argument set.

3.3 Specifying Types with Exceptions

Instead of specifying operations with non-trivial pre-conditions or arbitrarily defining a result, one often wishes a program operation to signal an *exception* [Goo75]. A programming language can define a mechanism to handle exceptions that arise while executing an invocation, as is done in CLU [LS79] and Trellis/Owl.

For simplicity, whenever exceptions are discussed, all operations of an algebra are thought of as returning `OneOf` objects.

A `OneOf` type is like the variant or discriminated union types that appear in some programming languages, such as CLU [LAB+81]. The carrier set and specification functions for `OneOf` types are defined as shown by the example trait "OneOf[normal: Int, empty: Null]," which is found in Figure 3.4. (The type `Null` has only one proper object, denoted by the result of the specification function "nil"; it is used as a placeholder in `OneOf` types. The type `Tag` contains proper objects for each possible `OneOf` tag.) The program operation symbols for a `OneOf` type are similarly defined.

To explain the syntactic sugar for specifications that use exceptions, consider the specification of the type `IntSet2` given in Figure 3.5.

Each program operation specification is rewritten so that it returns a `OneOf` type instead. A `OneOf` object with tag `normal` models the normal return. Each exception result is denoted by a `OneOf` object with a tag that is the same as the exception's name. (The name `normal` is not allowed as an exception name.) The rewriting is straight-forward. As an example, the specification of `choose` in `IntSet2` is syntactic sugar for the specification of Figure 3.6.

All `OneOf` types used in this fashion have their specification implicitly included in a specification that uses them.

One can also specify an operation so that it has a choice between signalling and returning a normal result. (An example is given in Figure 4.6.)

3.4 Function Specifications

Functions that are parts of programs are specified in much the same way as the program operations of a type. However, function specifications are generic in the sense that they may be interpreted using different

OneOf[normal: Int, empty: Null]: **trait**
introduces
make_normal: Int → NE
make_empty: Null → NE
hasTag?: NE, Tag → Bool
#==#: NE, NE → Bool
val_normal: NE → Int
val_empty: NE → Null
asserts for all [i: Int, o_1, o_2: NE]
hasTag?(make_normal(i), normal) = true
hasTag?(make_empty(nil), empty) = true
hasTag?(make_empty(nil), normal) = false
hasTag?(make_normal(i), empty) = false
val_normal(make_normal(i)) = i
val_empty(make_empty(nil)) = nil
(o_1 == o_2) = (o_1 = o_2)
exempts for all [i: Int]
val_normal(make_empty(nil))
val_empty(make_normal(i))

Figure 3.4: The trait OneOf[normal: Int, empty: Null].

IntSet2 **immutable type**
class ops [null]
instance ops [ins, elem, choose, size, remove]
based on sort C **from** IntSetTrait

op null(c:IntSetClass) **returns**(s:IntSet)
ensures s == {}

op ins(s:IntSet, i:Int) **returns**(r:IntSet)
ensures r == s ∪ {i}
op elem(s:IntSet, i:Int) **returns**(b:Bool)
ensures b = (i ∈ s)
op choose(s:IntSet2) **returns**(i:Int)
signals(empty(Null))
ensures (isEmpty(s) ⇒
signals empty(nil(Null)))
& ((¬ isEmpty(s)) ⇒ i ∈ s)
op size(s:IntSet) **returns**(i:Int)
ensures i = toInt(size(s))
op remove(s:IntSet, i:Int) **returns**(r:IntSet)
ensures r == delete(s,i)

Figure 3.5: Specification of the type `IntSet2`.

op choose(s:IntSet2) **returns**(o: OneOf[normal: Int,
empty: Null])
ensures (isEmpty(s) ⇒
o == make_empty(nil))
& ((¬ isEmpty(s)) ⇒
(hasTag?(o,normal)
& val_normal(o) ∈ s))

Figure 3.6: Desugared form of an exception specification.

⟨function specification⟩ ::= **fun** ⟨nominal signature⟩
⟨pre-condition⟩ ⟨post-condition⟩

Figure 3.7: Syntax of Function Specifications.

sets of type specifications, which allows one to add new types to a program without editing function specifications.

The syntax of function specifications is similar to that for program operation specifications except that one uses **fun** instead of **op** (see Figure 3.7). An example, the specification of *is2in*, is given in Figure 3.8. The pre-condition of *is2in* is "true" and the post-condition is "b = (2 ∈ s)." Like program operation specifications, the pre-conditions and post-conditions of function specifications must be subtype-constraining. An omitted pre-condition is sugar for "true".

The assertions in the pre- and post-conditions of a function specification may use specification functions from the specifications of any of the types mentioned in the nominal signature. The collection of the ⟨type spec⟩s for the non-visible types mentioned, either directly or indirectly. in the nominal signature of a function specification is called the *base specification set* of the function specification. For example, the base specification set of *is2in* includes only `IntSet` (as `Bool` is a visible type and thus need not be included). The assertions found in the pre- and post-conditions of a function specification must be $SIG(SPEC)$-assertions, where $SPEC$ is the base specification set.

The definition of satisfaction for function specifications is dependent on a definition of function implementations. That is, one needs a formal model of function implementations (the semantics of code written in

fun is2in(s:IntSet) **returns**(b:Bool)
requires true
ensures b = (2 ∈ s)

Figure 3.8: The function specification *is2in*.

a programming language) to describe satisfaction.

To manipulate objects abstractly, a function implementation must have access to the operations of an algebra. Hence function implementations take an algebra as an argument and then a tuple of arguments from the carrier set of that algebra, producing an set of objects in the carrier set of that algebra (representing the set of possible results). Furthermore, the algebra that is an argument to the function implementation must be such that the base specification set's signature is a subsignature of the algebra's.

Definition 3.4.1 (satisfies for functions). Let S_f be the following function specification:

fun f($\vec{\mathtt{x}} : \vec{\mathtt{S}}$) **returns**($\mathtt{v} : \mathtt{T}$)
requires R
ensures Q.

Let Σ' be the signature of the base specification set of S_f. Let *SPEC* be a set of type specifications that includes the base specification set and such that Σ' is a subsignature of $SIG(SPEC)$. Let the presumed subtype relation of *SPEC* be $\leq$. Let $X = \{\mathtt{x}_1 : \mathtt{S}_1, \ldots, \mathtt{x}_n : \mathtt{S}_n\}$. A function f *satisfies* S_f *with respect to SPEC* if and only if for all *SPEC*-algebras C, for all proper $SIG(SPEC)$-environments $\eta_C : X \rightarrow |C|$, the following condition holds. If

$$(C, \eta_C) \models R, \tag{3.14}$$

then for all possible results $q \in f(C)(\eta_C(\vec{\mathtt{x}}))$

$$q \neq \perp \tag{3.15}$$

$$(C, \eta_C[q/\mathtt{v}]) \models Q, \tag{3.16}$$

and there is some $\mathtt{U} \in TYPES$ such that $q \in \mathtt{U}^C$ and $\mathtt{U} \leq \mathtt{T}$. Furthermore, whenever some argument to the operation is $\perp$, then the only possible result is $\perp$.

In the above definition, notice that the possible results must have some type that is a presumed subtype of the nominal result type. This ensures that when an identifier of the nominal result type is bound to a possible result, that binding obeys the presumed subtype relation.

The phrase "f satisfies S_f" will be used instead of "f satisfies S_f with respect to *SPEC*" when *SPEC* is clear from context.

Example 3.4.2. As an example of function satisfaction, consider the following function implementation:

$$f \stackrel{\text{def}}{=} \lambda A.\lambda s.\mathtt{elem}^A(s, 2). \tag{3.17}$$

This function satisfies the specification *is2in* given above with respect to II. To see this, let C be a II-algebra, and let $\eta_C : \{\mathtt{s} : \mathtt{IntSet}\} \rightarrow |C|$ be an environment such that $\eta_C(\mathtt{s}) = \{1, 2, 3\}$. Then $(C, \eta_C) \models$ true, so the pre-condition is satisfied. Let $r \in \mathtt{elem}^C(\eta_C(s), 2)$ be a possible result. Then r is proper, has type `Bool` and is such that

$$(C, \eta_C[r/\mathtt{b}]) \models \mathrm{b} = (2 \in \mathrm{s}). \tag{3.18}$$

This last equation follows from II.

3.5 Discussion

The discussion that follows treats a limitation of the type system used in our specifications and various pitfalls for the user of the specification language defined above. Specifiers must be careful of how they specify specification functions for subtypes and how they use equality in specifications.

3.5.1 Loss of Information for Subtype Results

When the subtype's abstract values have more information than the supertype's, one has to be careful to write each specification so as not to lose information. However, the loss of information cannot be prevented in general for function specifications.

For example, consider a type `PSchd` of priority "schedulers"; this type will be a subtype of `IntSet`, with jobs represented by integers, and a `choose` operation that returns either the least or the greatest job number, depending on a "priority" that is set when the scheduler is created. The specification of this type is given in Figure 3.9. The abstract values of the type `PSchd` (see Figure 3.11) have more information than the abstract values of the type `IntSet`, namely a priority.

Consider the specification of the `ins` operation of `PSchd`. Because the behavior of `ins` is specified for `PSchd` arguments separately from the behavior for `IntSet` arguments, the implementation must leave the priority of a `PSchd` argument unchanged. Thus there is no loss of information.

However, for functions there is only one specification, so loss of information is inevitable, because of the restriction to subtype-constraining assertions. Consider the specification *ins3* of Figure 3.12. Since the argument is nominally an `IntSet`, there is no way to refer to the priority of the argument or the result. Since the assertion in the post-condition must be subtype-constraining, there is no way to say that the result must be the same as the result of the "$\cup$" specification function, which would preserve the priority of a `PSchd`. That is, the post-condition cannot be "w3 = s $\cup$ {3}", because this would be using equality (=) between terms that are not of a visible type. Hence implementations of *ins3* with respect to `IntSet` + `PSchd` need not leave the priority of a priority scheduler argument unchanged.

A solution to this problem would be to provide multiple specifications of functions, much as multiple specifications of program operations are provided in type specifications. An implementation would be required to satisfy the most specific applicable function specification, much the same as for program operation specifications. This would probably pose no great problems for reasoning. However, such an extension is left for future work.

A different approach to these problems in the context of record types is illustrated in the work of Jategaonkar and Mitchell on ML [JM88] and Cardelli and Mitchell [CM89].

3.5.2 Loss of Type Information for Subtype Results

The specification language cannot express certain constraints on the type of the value returned by a function. To illustrate the limitations, consider again the

PSchd **immutable type**
subtype of IntSet **by** $\langle b, s \rangle$**simulates**s
class ops [new]
instance ops [ins, elem, choose, size, remove, leastFirst]
based on sort C **from** PSchdTrait

op new(c:PSchdClass, b:Bool) **returns**(p:PSchd)
ensures p == ⟨b, {}⟩

op ins(p:PSchd, i:Int) **returns**(r:PSchd)
ensures (r == insert(p, i))
& (r.first = p.first)
op elem(p:PSchd, i:Int) **returns**(b:Bool)
ensures b = i ∈ p
op choose(p:PSchd) **returns**(i:Int)
requires ¬ isEmpty(p)
ensures i ∈ p
& (p.first ⇒ lowerBound?(p.second,i))
& ((¬p.first)
⇒ upperBound?(p.second,i))
op size(p:PSchd) **returns**(i:Int)
ensures i = toInt(size(p))
op remove(p:PSchd, i:Int) **returns**(r:PSchd)
ensures (r == delete(p,i))
& (r.first = p.first)
op leastFirst(p:PSchd) **returns**(b:Bool)
ensures b = p.first

Figure 3.9: Specification of the priority scheduler type, `PSchd`.

OrderedIntSet: **trait**
imports IntSetTrait
assumes Ordered **with** [Int **for** T]
introduces
lowerBound?: C, Int → Bool
upperBound?: C, Int → Bool
asserts for all [s: C, i, j: Int]
lowerBound?({},i) = true
lowerBound?(insert(s,j),i)
= (($i \leq j$) & lowerBound?(s,i))
upperBound?({},i) = true
upperBound?(insert(s,j),i)
= (($i \geq j$) & upperBound?(s,i))

Figure 3.10: The trait OrderedIntSet.

PSchdTrait: **trait**
imports OrderedIntSet,
Pair **with** [Bool **for** T1, IntSet **for** T2]
introduces
toSet: C → IntSet
size: C → Card
insert,delete: C,Int → C
#∈#: Int,C → Bool
isEmpty: C → Bool
∪, ∩: C,C → IntSet
∪, ∩: C,IntSet → C
∪, ∩: IntSet,C → C
#==#: C,C → Bool
#==#: C,IntSet → Bool
#==#: IntSet,C → Bool
asserts
for all [p, p_2: C, s: Set, b: Bool, i, j: Int]
toSet(p) = p.second
size(p) = size(toSet(p))
($i \in p$) = ($i \in$ toSet(p))
insert(p, i) = ⟨ p.first, insert(toSet(p), i) ⟩
delete(p, i) = ⟨ p.first, delete(toSet(p), i) ⟩
isEmpty(p) = isEmpty(toSet(p))
(p == p_2) = (toSet(p) = toSet(p_2))
(p == s) = (s == toSet(p))
(s == p) = (s == toSet(p))
($p \cap p_2$) = (toSet(p) ∩ toSet(p_2))
($p \cap s$) = (⟨ p.first, toSet(p) ∩ s ⟩)
($s \cap p$) = (⟨ p.first, toSet(p) ∩ s ⟩)
($p \cup p_2$) = (toSet(p) ∪ toSet(p_2))
($p \cup s$) = (⟨ p.first, toSet(p) ∪ s ⟩)
($s \cup p$) = (⟨ p.first, toSet(p) ∪ s ⟩)

Figure 3.11: The trait PSchdTrait.

fun ins3(s:IntSet) **returns**(w3:IntSet)
ensures w3 == s ∪ {3}

Figure 3.12: Specification of the function *ins3*, which inserts 3 in a set.

function specification *ins3*, in Figure 3.12. An obvious implementation is the following function:

$$ins3 \stackrel{\text{def}}{=} \lambda A.\lambda s.\mathtt{ins}^A(s, 3). \qquad (3.19)$$

Notice that if one passes this implementation an object of type `PSchd`, one gets back an object of type `PSchd`. However, the type system of the specification language cannot express this. One way to do so would be to use a kind of bounded quantification [CW85]. For example one might specify *ins3* with the following heading:

fun ins3(s:t $\leq$ IntSet) **returns**(w3:t)

The use of bounded quantification solves the problem with the type of the result, since one can conclude that if *ins3* is passed an instance of `PSchd`, then it returns an instance of `PSchd`. However, such a function specification cannot be satisfied with respect to arbitrary sets of type specifications. For example, consider a subtype of `IntSet` whose objects cannot contain 3. The problem is that the returned object cannot be an element of such a type. The problem is similar to the semantic problems with bounded quantification pointed out by [BL88].

A better solution to this problem would be to allow multiple specifications for functions, as described above. A different approach would be to use the notion of "F-bounded quantification" [CCH+89].

3.5.3 The Need for Subtype-Constraining Assertions

In the specification of `IntSet` given in Figure 1.1, the pre-condition of the `choose` operation is written as "¬ isEmpty(s)" instead of "¬(s = {})." However, for the soundness of program verification (see Chapter 6) the assertions used in specifications cannot use equality (=) between terms of nonvisible sorts; such assertions are called subtype-constraining. The term "¬(s = {})" does not restrict subtypes as well as "¬ isEmpty(s)," since subtypes such as `PSchd` may use different abstract values and so each element of the subtype will satisfy the pre-condition trivially. Hence, even though these two terms are logically equivalent for sets, they are not equivalent when used in a specification. This problem is much like those that arise when reasoning about nonstandard models of sets. The problem is avoided by requiring that assertions in specifications be subtype constraining.

In [Lea89], the requirements on assertions were even stronger, as one had to use "program observable" assertions in specifications. The requirement of program observability is more severe, since it may restrict how specifications can be written. These problems are illustrated by the following example, suggested by W. Weihl (personal communication, 1989).

Consider a statistical database type, as specified in Figures 3.13 and 3.14. The assertion that describes the post-condition of the `ins` operation is not program observable in the sense that there is no program one can write that will return "true" only if the assertion "q == insert(s, r)" is true, given that one can only observe objects of type `Sdb` by calling the operations `ins`, `mean`, and `sampleVariance`. That is, there are several multi-sets that have the same mean and variance, but which are not equal to the one required. However, the following assertion is program observable:

Sdb **immutable type**
class ops [new]
instance ops [ins, mean, sampleVariance]
based on sort C **from** StatBag

op new(c:SdbClass) **returns**(s:Sdb)
ensures s == {}

op ins(s:Sdb, r:Rat) **returns**(q:Sdb)
ensures q == insert(s, r)

op mean(s:Sdb) **returns**(r:Rat)
requires ¬ isEmpty(s)
ensures r == (sum(s)/size(s))

op sampleVariance(s:Sdb) **returns**(r:Rat)
requires count(s) > 1
ensures r == variance(s)

Figure 3.13: Specification of the statistical database type, `Sdb`.

```
((sum(q) / size(q))
        == (sum(insert(s,r))/size(insert(s,r))))
    & (variance(q) == variance(insert(s,r))).
```

The above assertion states the required properties, but it is less terse.

Assertions that are subtype-constraining but not program observable are often used in such situations where a slight overspecification is preferable to such long assertions.

A way to make the assertion "q == insert(s,r)" program observable is to add more operations to the type `Sdb`. However there may be other reasons, such as modularity or the privacy of information maintained by a program, that would prohibit a designer from choosing to add such operations. Another way would be to describe the specification function "==" so that it related multi-sets with the same mean and variance. However, that would make the trait StatBag less intuitive.

Therefore the advantage of the semantics of assertions given above is that the restriction to subtype-constraining assertions allows more freedom to the specifier. Subtype-constraining assertions such as "q == insert(s,r)" can be used in specifications, even though they are not program observable.

3.5.4 No Constraints Imposed on the Operations of Presumed Subtypes

The only constraints imposed on the operations of a subtype in this chapter are syntactic. That is, the semantic constraints imposed in Chapter 4 do not affect the construction of algebraic models. A set of type specifications has a meaning regardless of whether the presumed subtype relation is a subtype relation. This separation is achieved because the behavior of a pro-

StatBag: **trait**
imports Rational
BagBasics **with** [Rat **for** E]
introduces
#==#: C,C → Bool
sum,variance: C → Rat
asserts for all [b_1, b_2: C, i, m: Rat]
(b_1 == b_2) = (b_1 = b_2)
sum({}) = 0
sum(insert(b_1, i)) = i + sum(b_1)
mean(b_1) = (sum(b_1)/size(b_1))
sd2({},m) = 0
sd2(insert(b_1, i), m)
= (sd2(b_1,m) + (($i - m$)*($i - m$)))
variance(b_1)
= (sd2(b_1,mean(b_1))/(size(b_1)-1))
exempts for all [r: Rat]
mean({})
variance({})
variance(insert({},r))

Figure 3.14: The trait StatBag.

gram operation of an algebraic model only has to satisfy the program operation specification that has the most specific applicable argument type requirements.

The benefit of having meaningful type specifications even when the presumed subtype relation does not satisfy the semantic requirements for a subtype relation is that one does not have to consult a presumed supertype's operation specification to discover the meaning of a subtype's operation specification. For example, one might have a specification language where the pre-condition of a subtype's operation was a disjunction of the specificed pre-condition and the pre-conditions of its supertypes and where the post-condition of a subtype's operation was a conjunction of the specified post-condition and the post-conditions of its supertypes[1]. However, in that case it would be easy to specify a subtype that could not be implemented, because the post-condition of the subtype might contradict the post-condition of the supertype. Furthermore, the contradiction would not be obvious, as it would require looking at the specification of all supertypes.

3.5.5 Inheritance of Specifications

A better way to abbreviate operation specifications is to omit them entirely. That is, one can have inheritance of operation *specifications*. The semantics of the specification language does not prohibit inheritance of operation specifications, since an operation specification applies to all argument combinations for which there is no more specific operation specification. Indeed, one would only have to respecify an operation in a subtype if one wanted to make the specification more specific (strengthening the post-condition or naming different argument or result types).

To make specifications easier to read, it would perhaps be best to explicitly list the instance operation names that one wanted to be inherited from the supertype specification.

[1] as has been suggested by B. Meyer for Eiffel on the newsgroup comp.lang.eiffel

op assertEmpty(s:IntSet) **returns**(b:Bool)
requires isEmpty(s)
ensures b = true.

Figure 3.15: Specification of the program operation `assertEmpty`.

3.5.6 Must Subtypes Implement All Instance Operations of Supertypes?

It is nearly dogma that a subtype must have all the instance operations of its supertypes, as required by the definition of signatures given in Chapter 2. However, what would happen if the operation specification of Figure 3.15 were added to the specification of the type `IntSet` as an additional instance operation specification? The operation `assertEmpty` need not be implemented for the type `Interval`, since there are no empty intervals, and hence no `Interval` arguments could reasonably be sent that message. Such an operation should not even have to be specified in the specification of a subtype, although technically in the algebraic models $ResSort(\texttt{assertEmpty}, \langle\texttt{Interval}\rangle)$ would be defined. This example shows that the common notion that the subtype must implement all the instance operations of a supertype is not necessary.

However, such a radical departure from accepted practice may not be wise. That is, it is valuable to have syntactic checks for the plausibility of specified subtype relations, and one of the most basic of such checks is that the subtype should have all the instance operations of the supertype.

Chapter 4

Subtype Relations

The formal definition of subtype relations is presented in this chapter, as well as a discussion of issues, examples, and related work.

The definition of subtype relations allows modular reasoning about object-oriented programs that use subtype polymorphism, as described in Chapter 1. This style of reasoning relies on nominal type information; that is, the types declared for formal arguments and results. The use on nominal type information in specifications is discussed above (in Chapter 3). The method for program verification described in Chapter 6, allows one to reason about the result of an expression with nominal type `T` as if each possible result was an instance of type `T`. For example, to verify that the function

```
fun is2in(s: IntSet): Bool = elem(s,2)
```

implements the specification of Figure 3.8, one is allowed to assume that `s` denotes instances of `IntSet`, even though one can pass instances of subtypes of `IntSet` (such as `Interval`) as actual arguments to *is2in*.

The semantic property that characterizes a subtype relation is the existence of a simulation. For example, each instance of type `Interval` simulates an instance of `IntSet` with the same elements (for some implementation of `IntSet`). Together with a type system that ensures that the possible results of each expression of some nominal type `IntSet` are instances of some subtype of `IntSet`, this property ensures the soundness of the verification method presented in Chapter 6. Informally, soundness holds because if one verifies some property of an expression of nominal type `IntSet`, then the expression can only denote an object of some subtype of `IntSet`, which must simulate an object of type `IntSet`, hence (provided the notion of simulation is adequate), the behavior of the object denoted by the expression will not be surprising.

Subtype relations are defined among first-order, immutable, *abstract* types as characterized by type specifications — not just the built-in types of a particular programming language. Thus one can specify subtype relations among abstract types, such as nonempty sets of integers or queues of length ten. In general, an abstract type can be thought of as a set of values that satisfy some semantic property. These properties can be used in reasoning about objects of that type. By contrast, a notion of subtyping that is limited to built-in types does not allow one to use semantic properties (such as being nonempty or of length exactly ten) in reasoning.

4.1 Definition of Subtype Relations

The formal definition of subtype relations is parameterized by the semantics of a specification. This allows the definition of subtype relations to be independent of the form of specifications and particular specification languages. Furthermore, since the semantics of a specification is a set of algebraic models, the definition handles incompletely specified types. Such specifications are important because they leave room for implementation decisions and later specialization of subtypes.

Definition 4.1.1 (subtype relation). Let Σ be a signature. Let $SPEC$ be a nonempty collection of Σ-algebras with the same $SIG(B)$-reduct, where B is a fixed algebra that defines the visible types. Let $\leq$ be the presumed subtype relation of Σ. Then $\leq$ is a *subtype relation on the types of* $SPEC$ if and only if for all algebras $C \in SPEC$, there is some $A \in SPEC$ such that there is a Σ-simulation relation between C and A.

The specification language presented in Chapter 3 has one define a simulation relation when giving a type specification; if one does this correctly, then one can be sure that the required simulation relation always exists. Since the set of algebras with the same signature is the meaning of a set of type specifications $SPEC$, in what follows the phrase "subtype relation on the types of $SPEC$" will also be used to mean a subtype relation on the types of the set of algebras that are the meaning of $SPEC$.

The requirement that a Σ-simulation relation exist does test properties of the presumed subtype relation $\leq$ (which is a component of Σ), although it may not be apparent from the above definition. This is because a family of relations can only be a Σ-simulation relation if it has various properties related to $\leq$, such as the ability to coerce elements of subtypes to supertypes. Most importantly, the substitution property also depends on the signature Σ, which is used, for example, in determining nominal sorts and types.

A trivial example of a subtype relation is the identity relation on types; the identity relation is always a subtype relation because simulation is reflexive.

More interesting examples of subtype relations are considered below: first among nondeterministic types, and then incompletely specified types.

4.1.1 Subtypes can be More Deterministic

A subtype can be more deterministic than its supertypes. An example is that `Interval` is a subtype of `IntSet` in the specification II that combines

them both. Recall that the `choose` operation, when applied to a nonempty `IntSet`, is allowed to return any integer in the `IntSet`. The result of applying `choose` to an empty `IntSet` is undefined. The result of applying `choose` to an `Interval` is its least element. The presumed subtype relation of II, $\leq$, is the smallest reflexive relation on the types of II such that `Interval` $\leq$ `IntSet`.

Let C be an II-algebra. Let A be an algebra that is the same as C, except that its `choose` operation exhibits all the nondeterminism allowed by its specification. Then A is an II-algebra. One must be able to pick an algebra other than C, because the operations of C may not be nondeterministic enough for the simulation to preserve the behavior of the `choose` operation.

The simulation relation $\mathcal{R}$ between C and A is constructed from the specification of II, using the specified **simulates** relationships as a guide. The proper elements of carrier sets of each sort `T` in both algebras are generated by the specification functions, and hence by ground terms of sort `T`. Therefore, for each type `T`, and for each ground term m of sort `T`, let $\mathcal{R}_{\mathtt{T}}$ relate each the value of m in C to the value of m in A and $\perp$ to $\perp$. Also for each sort `T` with presumed subtypes, add the relationships specified in the subtype clauses. In general these clauses are given using terms containing identifiers, which are presumed to be related. The denotations of each term are thus to be related by $\mathcal{R}$. In the specification of `Interval` the subtype clause is the following

subtype of IntSet **by** $[l, u]$ **simulates** toSet($[l, u]$)

(The specification function "toSet" is specified in Figure 3.3.) This clause specifies the following relationships

$$[l, u] \, \mathcal{R}_{\mathtt{IntSet}} \, \text{toSet}([l, u]). \tag{4.1}$$

So for all environments η_C and η_A that are defined on variables l, and u of nominal type `Int` and such that

$$\eta_C(l) \quad \mathcal{R}_{\mathtt{Int}} \quad \eta_A(l) \tag{4.2}$$
$$\eta_C(u) \quad \mathcal{R}_{\mathtt{Int}} \quad \eta_A(u) \tag{4.3}$$

one must ensure that $\mathcal{R}_{\mathtt{IntSet}}$ contains the relationship:

$$\overline{\eta_C}[\![[l, u]]\!] \, \mathcal{R}_{\mathtt{IntSet}} \, \overline{\eta_A}[\![\text{toSet}([l, u])]\!]. \tag{4.4}$$

The constructed relationship relates `Interval` objects to `IntSet` objects with the same elements. Finally, if a sort `T` has presumed subtypes, then add each relationship that holds at a presumed subtype to $\mathcal{R}_{\mathtt{T}}$ (this ensures that $\mathcal{R}_{\mathtt{T}} \supseteq \bigcup_{\mathtt{S} \leq \mathtt{T}} \mathcal{R}_{\mathtt{S}}$).

It is not difficult to show that the family of relations $\mathcal{R}$ as constructed above is a simulation relation. For example, to show the substitution property for the program operation `choose`, suppose $q \, \mathcal{R}_{\mathtt{IntSet}} \, r$; then a possible result of $\mathtt{choose}^C(q)$ must be a possible result of $\mathtt{choose}^A(r)$, because q and r have the same elements and A's `choose` operation is maximally nondeterministic. (More details are presented in Example 2.2.2.) Hence the presumed subtype relation of II is a subtype relation.

op choose(s:Crowd) **returns**(i:Int)
requires $\neg$ isEmpty(s)
ensures i = choice(s)

Figure 4.1: Specification of the choose operation of the type `Crowd`.

As a counterexample, it is easy to show that `IntSet` cannot be a subtype of `Interval`, because the `choose` operation applied to a `IntSet` object may have more possible results than the `choose` operation applied to a `Interval` object. That is, if `v` has nominal type `Interval`, then one can conclude from the specification of `Interval` that

```
equal(choose(v),choose(v))
```

will always return *true*. However, if `IntSet` were a subtype of `Interval`, then the above conclusion would not be valid.

4.1.2 Incompletely Specified Supertypes

The definition of subtype relations says more than the intuition that "each object of the subtype simulates some object of the supertype." In this sub-section, some examples of incompletely specified types are discussed that clarify how subtyping interacts with incomplete specifications.

The specification of the type `IntSet` is *incomplete*, since some algebras that satisfy that specification are not observably equivalent. (That is, a program that observed one `IntSet`-algebra might have a different set of possible results than a program that observed another algebra.)

Although the type `IntSet` has maximally nondeterministic models that capture all the behavior of the specification, there are types for which no such models exist. For example, consider the type `Crowd` that is the same as `IntSet`, except that its `choose` operation, when applied to a nonempty `Crowd` object, is required to be deterministic. This can be specified by making the post-condition of `choose` return a result that is the result of an incompletely specified specification function, as in Figure 4.1.

The type `Crowd` can be a subtype of `IntSet`, but not vice versa. The reason that `Crowd` is a subtype of `IntSet` is that `Crowd`'s `choose` operation is more deterministic, hence its results will not be surprising.

The type `Interval` (with suitable changes) can be considered to be a subtype of `Crowd`, because there are algebraic models of `Crowd` + `Interval` such that the `choose` operation always returns the least element of a `Crowd` object, as does the `choose` operation when applied to an `Interval`. But `Crowd` is not a subtype of `Interval`, because if one takes a model such that the `choose` operation returns the greatest element of a `Crowd` object, then there would be no model that one could find such that the elements of the carrier set of `Crowd` simulated the elements of the carrier set of `Interval` (because the choose operation when applied to an `Interval` always returns the least element, regardless of the model).

The most subtle (lack of a) relationship is between PSchd and Crowd. Clearly, Crowd cannot be a subtype of PSchd, because the choose operation of Crowd does not have to always return either the greatest or the least element (it might, for example, return the middlemost element). However, one might think that PSchd could be a subtype of Crowd, because each PSchd simulates some Crowd object; a least-first instance of PSchd simulates a Crowd object in a model of Crowd where choose returns the least element, and similarly greatest-first instances of PSchd simulate Crowd objects in other models. But, according to the definition of subtype relations, all the objects in a given model of PSchd would have to simulate some Crowd object, and all of these Crowd objects would have to be in a single model's carrier set. Since the choose operation cannot return both the least and the greatest element of a Crowd, since the simulation must be among objects with the same elements, and since the carrier set of Crowd is generated by the specification functions, there can be no such simulation.

The fact that PSchd cannot be a subtype of Crowd can be proved as follows. Let CP be a specification that combines the types PSchd and Crowd, with PSchd as a presumed subtype of Crowd (the trait for PSchd would have to be respecified, but this is simple because Crowd is so much like IntSet). Let C be a CP-algebra. For the sake of contradiction, suppose that the CP-algebra demanded by the definition of subtype relations is A. Clearly, to preserve the meaning of the program operations, an empty PSchd object can only simulate an empty Crowd object. Note that there are two kinds of empty PSchd objects, those such that leastFirst returns *true* and those for which it returns *false*. Since the type PSchd is generated by the specification function $\langle \#, \# \rangle$, we can write these empty PSchd objects in C as $\langle true, \{\} \rangle$, and $\langle false, \{\} \rangle$. Since the type Crowd is generated by the specification functions $\{\}$ and "insert" and is partitioned by $\in$ [GH86b], there can only be one empty Crowd object in A, which can be denoted by $\{\}$. Then by the substitution property, one would have:

$$\texttt{ins}^C(\langle true, \{\} \rangle, 1) \quad \mathcal{R}_{\texttt{Crowd}} \quad \texttt{ins}^A(\{\}, 1) \tag{4.5}$$

$$\texttt{ins}^C(\langle false, \{\} \rangle, 1) \quad \mathcal{R}_{\texttt{Crowd}} \quad \texttt{ins}^A(\{\}, 1) \tag{4.6}$$

The substitution property can be applied again to yield:

$$\texttt{ins}^C(\texttt{ins}^C(\langle true, \{\} \rangle, 1), 2) \quad \mathcal{R}_{\texttt{Crowd}} \quad \texttt{ins}^A(\texttt{ins}^A(\{\}, 1), 2) \tag{4.7}$$

$$\texttt{ins}^C(\texttt{ins}^C(\langle false, \{\} \rangle, 1), 2) \quad \mathcal{R}_{\texttt{Crowd}} \quad \texttt{ins}^A(\texttt{ins}^A(\{\}, 1), 2) \tag{4.8}$$

(Although technically the result of an operation is a set of possible results, the above relationships should be read as if the result was a single object, and the operations involved are deterministic in any case.) But the above leads to a contradiction, because the objects on the left-hand sides above, $\langle true, \{1, 2\} \rangle$ and $\langle false, \{1, 2\} \rangle$, behave differently on the choose operation, but the object on the right-hand sides ($\{1, 2\}$) can only have one response to the choose operation, since the choose operation of Crowd is deterministic. So PSchd cannot be a subtype of Crowd.

The contradiction that prevents a simulation in the counterexample above is relevant to reasoning about programs. For example, suppose one had a function that took two objects of type Crowd, checked that they were empty, inserted 1 and 2 in each (producing two new objects), and then called the choose operation on each. This programmer of this function could reasonably expect that the choose invocations would return the same integer, since choose is deterministic. However, if PSchd were allowed to be a subtype of Crowd, then one could pass two different, empty PSchd objects to this function such that the results of the choose operation would be different. This kind of surprising behavior is ruled out by the definition of subtype relations.

Although each individual instance of PSchd "acts like" some instance of Crowd, in general these instances of Crowd must be from different algebras. So a definition that simply required each object of a subtype to "act like" some object of a supertype would not be adequate to prevent the reasoning problems discussed above. These problems are only prevented by requiring each object of a subtype to simulate an object of the supertype in the same algebra. Since algebras are abstractions of implementations, this amounts to saying that each object of the subtype must simulate some object of the supertype in a *single* implementation of the supertype.

4.2 Examples

The examples discussed in this section are of interest for comparison with related work, making generalizations about semantic relationships between subtypes and supertypes, and studying the interaction between subtyping and exceptions.

4.2.1 OneOf Types

Oneof types are useful in modeling exceptions (see Chapter 3), and are also one of the earliest studied examples of subtyping [Car84]. Cardelli's rule for OneOf types is that a OneOf with fewer tags is a subtype of a OneOf type with more tags, provided the corresponding fields were in the subtype relation. The example below thus shows that each instance of Cardelli's rule is a subtype relation. (See [Lea89] for an a similar comparison with immutable record types.)

A specification of the type OneOf[normal:Int, empty:Null] is given in Figure 4.2, where the type name is abbreviated to NE. This abbreviation will also be used in the rest of this chapter. For each type T, there is an instance operation named value[T]. The trait used to define the carrier set and specification functions of this type is found in Figure 3.4. The specification of OneOf[normal:Int] in Figure 4.3 is similar. However, the trait for OneOf[normal:Int] in Figure 4.4 also describes the specification function "value_empty" that is defined for the supertype. One cannot use the same trait as for NE, because then there would be some elements in the carrier set that would be generated by the specification functions but not by the program operations.

There are other ways to specify OneOf types so that the type OneOf[normal:Int] is a subtype of the type OneOf[normal:Int, empty:Null]. Instead of

```
NE immutable type
    class ops [make_normal, make_empty]
        instance ops [hasTag?, value[T]]
    based on sort NE
            from OneOf[normal: Int, empty: Null]

    op make_normal(c:NEClass, i: Int)
            returns(o:NE)
        ensures o == make_normal(i)

    op make_empty(c:NEClass, n: Null)
            returns(o:NE)
        ensures o == make_empty(n)

    op hasTag?(o:NE, t: Tag) returns(b:Bool)
        ensures b = hasTag?(o, t)

    op value[T](o:NE, t: Tag) returns(r:T)
        requires hasTag?(o, t)
            & ((t == normal) ⇒ (Int = T))
            & ((t == empty) ⇒ (Null = T))
        ensures ((t == normal)
                ⇒ (r == val_normal(o)))
            & ((t == empty)
                ⇒ (r == val_empty(o)))
```

Figure 4.2: The `OneOf` type `OneOf[normal: Int, empty: Null]`, abbreviated `NE`.

```
NI immutable type
    subtype of OneOf[normal: Int, empty: Null]
        by make_normal(i)
        simulates make_normal(i)
    class ops [make_normal]
    instance ops [hasTag?, value[T]]
    based on sort NI
        from OneOf[normal: Int]

    op make_normal(c:NIClass, i: Int)
            returns(o:NI)
        ensures o == make_normal(i)

    op hasTag?(o:NI, t: Tag) returns(b:Bool)
        ensures b = hasTag?(o, t)

    op value[T](o:NI, t: Tag) returns(r:T)
        requires hasTag?(o, t)
            & ((t == normal) ⇒ (Int = T))
        ensures (t == normal)
            ⇒ (r == val_normal(o))
```

Figure 4.3: The type `OneOf[normal: Int]`, abbreviated `NI`.

```
OneOf[normal: Int]: trait
    imports OneOf[normal: Int, empty: Null]
    introduces
        make_normal: Int → NI
        hasTag?: NI, Tag → Bool
        #==#: NI, NI → Bool
        #==#: NE, NI → Bool
        #==#: NI, NE → Bool
        val_normal: NI → Int
        val_empty: NI → Null
        toNE: NI → NE
    asserts for all [i: Int, o1, o2: NI, o3:NE]
        hasTag?(make_normal(i), normal) = true
        val_normal(make_normal(i)) = i
        (o1 == o2) = (toNE(o1) == toNE(o2))
        (o1 == o3) = (toNE(o1) == o3)
        (o3 == o1) = (toNE(o1) == o3)
        toNE(make_normal(i)) = make_normal(i)
    exempts for all [i: Int]
        val_empty(make_normal(i))
```

Figure 4.4: The trait OneOf[normal: Int].

operations that observe instances, one could have a programming language with built-in expressions for observing `OneOf` instances [LAB+81, Section 11.6] [CW85] [Car84]; however, it seems best to keep the type specification independent of the programming language by making the programming language's expressions syntactic sugar for operation invocations. One could also could specify `value_ni` operations for each tag `ni`; the pre-condition would then state when these operations were defined.

Let NEN be a specification that combines the two `OneOf` types discussed above. Let $\leq$ be the smallest reflexive relation on the types of NEN such that

$$\texttt{OneOf[normal : Int]} \quad \leq \quad \texttt{NE}.$$

Let C be an NEN-algebra and let A be an NEN-algebra that is the same as C except that the `value[T]` operations are maximally nondeterministic. Let $\mathcal{R}$ be the family of relations constructed according to the specification NEN. By construction, if q and r are proper and $q\ \mathcal{R}_{\texttt{NE}}\ r$, then q and r both have the same tag and the same value. Hence one can show that the relation $\mathcal{R}_{\texttt{NE}}$ is preserved by the specification functions and by the program operations. For example, if $q\ \mathcal{R}_{\texttt{NE}}\ r$, then `hasTag?` returns the same value on each (since they both have the same tag), and each possible result of $\texttt{value[Int]}^C(q, \texttt{normal})$ is a possible result of $\texttt{value[Int]}^A(r, \texttt{normal})$, because if the tag of q is `normal`, then the tag of r is `normal` and they have the same value; otherwise, if the tag of q is not `normal`, then since A is maximally nondeterministic, $\texttt{value[Int]}^A(r, \texttt{normal})$ has as its set of possible results all elements of the carrier set of `Int`. So $\leq$ is a subtype relation on NEN.

As a counterexample, note that the type `NE` cannot be a subtype of `OneOf[normal:Int]`. To see this, it suffices to note that the `hasTag?` operation when applied to an object of type `OneOf[normal:Int]` and the tag `normal` can only return *true*, but this property is not preserved by the presumed subtype `NE`.

4.2.2 Subtypes can have Weaker Requirements

The definition of subtype relations allows a subtype to be more defined than its supertypes, in the sense that pre-conditions of the subtype's instance operations may be weaker. For example, consider the type `PSchd2`, where `PSchd2` is exactly like `PSchd` except that the `choose` operation is specified as in Figure 4.5. This specification says that when the argument to `choose` is empty, the only possible result is zero. Let P2 be the specification that combines `PSchd` and `PSchd2`. Then the smallest reflexive relation $\leq$ on the types of P2 such that $\texttt{PSchd2} \leq \texttt{PSchd}$ is a subtype relation. This follows because if C is a P2-algebra, then there is a P2-algebra A that is the same as C except that the `choose` operation of A is maximally nondeterministic (when the pre-condition is not satisfied). Zero is the only possible result of `choose` on an empty `PSchd2` object, and zero is also a possible result of $\texttt{choose}^A$ when applied to an empty `PSchd` object. Hence the family of relations constructed from the specification's simulation specifications (in Figure 4.5) is a simulation relation.

PSchd2 **immutable type**
  subtype of PSchd **by** $\langle b, s\rangle$ **simulates** $\langle b, s\rangle$
  class ops [new]
  instance ops [ins, elem, choose, size, remove, leastFirst]
  based on sort C **from** PSchdTrait

  % other operations as in Figure 3.9.

  op choose(p:PSchd2) **returns**(i:Int)
    ensures (isEmpty(p) ⇒ i=0)
      & ((¬ isEmpty(p)) ⇒ i ∈ p.second)
      & (p.first ⇒ lowerBound?(p.second,i))
      & ((¬p.first)
        ⇒ upperBound?(p.second,i))

Figure 4.5: Specification of the type `PSchd2`, which is more defined than `PSchd`.

Allowing a subtype to be more defined seems right, since the idea of a pre-condition is to leave the behavior of an operation undefined when the pre-condition is not met.

4.2.3 Exceptions and Subtyping

Instead of specifying operations with nontrivial pre-conditions or arbitrarily defining a result, as was done for `PSchd2`, another way of dealing with boundary conditions is to specify that an operation should signal an exception.

In this subsection, operation specifications are again considered to be syntactic sugar for specifications of operations that return instances of a `OneOf` type, as in Section 3.3. For example, when the specification of the type `IntSet2` given in Figure 3.5 says that the `choose` operation signals "empty(nil)" when it is passed an empty instance of `IntSet2`. This means that `choose` returns an object of the type `NE` with tag `empty` and value `nil` when `choose` is passed an empty instance of `IntSet2`. Furthermore, `choose` returns an object of type `OneOf[normal:Int]` when passed an empty instance of `IntSet`.

Consider the specification `IntSet + IntSet2`. The type `IntSet2` is *not* a presumed subtype of `IntSet`. Informally, this is because `IntSet2`'s `choose` operation can signal an exception, which would be surprising if one thought that `choose` was being applied to a `IntSet`. More formally, consider the specified result sorts for the `choose` operation:

$$\begin{aligned} &\mathit{ResSort}(\texttt{choose}, \langle\texttt{IntSet}\rangle) \\ &\quad = \quad \texttt{OneOf[normal : Int]} \qquad (4.9) \\ &\mathit{ResSort}(\texttt{choose}, \langle\texttt{IntSet2}\rangle) \\ &\quad = \quad \texttt{OneOf[normal : Int, empty : Null]}. \qquad (4.10) \end{aligned}$$

If one was to have $\texttt{IntSet2} \leq \texttt{IntSet}$, then the mono-

IntSet3 **immutable type**
class ops [null]
instance ops [ins, elem, choose, size, remove]
based on sort C **from** IntSetTrait

% other operations as in IntSet.

op choose(s:IntSet3) **returns**(i:Int)
signals(empty(Null))
requires ¬ isEmpty(s)
ensures i ∈ s

Figure 4.6: Specification of the type `IntSet3`.

tonicity condition on signatures would require that:

$$\texttt{OneOf[normal : Int, empty : Null]} \leq \texttt{OneOf[normal : Int]}.$$

However, as noted above, this last relationship does not possess the required semantic properties, and thus such a relation $\leq$ would not be subtype relation. In other words, when one applies the `choose` operation to an empty `IntSet2` object, an object with tag `empty` is returned, whereas applying `choose` to an empty `IntSet` object can only return an object with the tag `normal`.

The reasoning above generalizes to arbitrary specifications. Thus a subtype's instance operation cannot have more exceptional results than its supertype's.

However, a subtype's operation can have fewer exceptional results than its supertype's if the supertype's specification allows a nondeterministic choice between signalling and returning normal results. For example, consider the type `IntSet3` as specified in Figure 4.6. The `choose` operation of `IntSet3` has a **requires** clause, like `IntSet`, but its signature allows the operation to signal, like the `choose` operation of `IntSet2`. Therefore, the `choose` operation of `IntSet3` can return any element of the carrier set of `OneOf[normal: Int, empty: Null]` when its precondition is not satisfied.

Consider the specification `IntSet2` + `IntSet3`, and assume that the specification of `IntSet2` states that `IntSet2` is a presumed subtype of `IntSet3`[1]. Let $\leq$ be the smallest reflexive relation on the types of this specification such that `IntSet2` $\leq$ `IntSet3`. It is easy to show that this $\leq$ is a subtype relation, since the subtype is more defined than its supertype.

It is also possible to show that the type `IntSet`, which has fewer exceptional results, is a subtype of `IntSet3`. Let II3 be a specification `IntSet` + `IntSet3`. The simulation between `IntSet` and `IntSet3` is specified as follows:

$$s \; \mathcal{R}_{\texttt{IntSet3}} \; s.$$

Let $\leq$ be the smallest reflexive relation on the types of II3 such that `IntSet` $\leq$ `IntSet3` and

$$\texttt{OneOf[normal : Int]} \leq \texttt{NE}.$$

Let C be an II3-algebra and let A be the same as C, except that its `choose` and `value[`T`]` operations are maximally nondeterministic. Then the family of relations $\mathcal{R}$ constructed from the specification is a simulation relation. By this construction, each object of type `IntSet` simulates an object of type `IntSet3` with the same elements. Suppose that $q \; \mathcal{R}_{\texttt{IntSet3}} \; r$; then

$$\texttt{choose}^C(q) \; \mathcal{R}_{\texttt{NE}} \; \texttt{choose}^A(r), \qquad (4.11)$$

because A is maximally nondeterministic. That is, if q is nonempty, so is r, and so each possible result of `choose` applied to q is a `OneOf` with tag `normal` and with a value that is an element of q and hence a possible result of `choose` applied to r. Furthermore, if q is empty, so is r; if q and r have the same type, then the possible results on q are included in those for r, since A is maximally nondeterministic; otherwise, if q has type `IntSet` and r has type `IntSet3`, then set of possible results of `choose` on r is the whole carrier set of the `OneOf` with tags `normal` and `empty`, and the possible results for q must be drawn from the carrier set of the `OneOf` with tag `normal`; hence each possible result for q simulates some possible result for r.

In general a subtype (such as `IntSet`) can have fewer exceptions than its supertypes (such as `IntSet3`), because a `OneOf` type with fewer tags can be a subtype of a `OneOf` type with the same tags and more.

4.2.4 Virtual Supertypes

In many practical examples of object-oriented design, one specifies types without class operations to be used as supertypes. Since these types do not have class operations they cannot be instantiated. A type that cannot be instantiated is called a *virtual type*, since its implementations often use virtual operations. A *virtual operation* has an implementation that uses some primitive (called **virtual** in Simula 67 [BDMN73]) to invoke an operation of a subclass[2]; hence a virtual operation cannot be executed unless the subclass has defined the required operation.

Consider the specification Vehicles, given in Figure 4.7. In this specification, `Vehicle` is a virtual type and has no class operations. The carrier set for `Vehicle` is described in the trait VehicleTrait found in Figure 4.8. (To specify a virtual type, one may be tempted to use a virtual trait — a trait without generators. However, if there was no "makeVehicle" specification function, then it would be difficult to describe how a bicycle simulates a vehicle.) The carrier set for `Bicycle` is described in the trait BicycleTrait found in Figure 4.9.

Let $\leq$ be the smallest reflexive relation on the types of Vehicles such that `Bicycle` $\leq$ `Vehicle`. Let C be a Vehicles-algebra. Then the family of relations $\mathcal{R}$ from C to itself constructed from the specification as

[1] In a practical specification language, it might be best to allow the supertype's specification to state the presumed subtype relationship, to allow for such "after the fact" supertypes.

[2] Some researchers (e.g., [SCW85, Page 42] [Sym84, Page 450]) call a type or class that cannot be instantiated "abstract," but this leads to confusion with the term "abstract data type." In Eiffel, operations that are not implemented are said to be "deferred" [Mey88].

Vehicle **virtual type**
instance ops [wheels, passengers]
based on sort C **from** VehicleTrait

op wheels(v:Vehicle) **returns**(i:Int)
ensures i = wheels(v)

op passengers(v:Vehicle) **returns**(i:Int)
ensures i = passengers(v)

Bicycle **immutable type**
subtype of Vehicle **by** s $\mathcal{R}$ makeVehicle(2,1)
class ops [new]
instance ops [wheels,passengers,maker]
based on sort C **from** BicycleTrait

op new(BicycleClass, s:String)
returns(b:Bicycle)
ensures b == makeBike(s)

op wheels(b:Bicycle) **returns**(i:Int)
ensures i = 2

op passengers(b:Bicycle) **returns**(i:Int)
ensures i = 1

op maker(b:Bicycle) **returns**(s:String)
ensures s == company(b)

Figure 4.7: The specification Vehicles, including types `Vehicle` and `Bicycle`.

VehicleTrait **trait**
introduces
makeVehicle: Int,Int → C
wheels: C → Int
passengers: C → Int
asserts for all [w, p: Int]
wheels(makeVehicle(w,p))
= **if** $w \geq 1$ **then** w **else** 1
passengers(makeVehicle(w,p))
= **if** $p \geq 1$ **then** p **else** 1

Figure 4.8: The trait VehicleTrait.

BicycleTrait **trait**
introduces
makeBike: String → C
wheels: C → Int
passengers: C → Int
company: C → String
#==#: C,C → Bool
asserts for all [s: String, c_1, c_2: C]
wheels(makeBike(s)) = 2
passengers(makeBike(s)) = 1
company(makeBike(s)) = s
(c_1 == c_2)
= (company(c_1) == company(c_2))

Figure 4.9: The trait BicycleTrait.

in Example 4.1.1 is a simulation relation. According to the specification, the relation $\mathcal{R}_{\texttt{Vehicle}}$ is such that each proper element of the carrier set of `Bicycle` is related to the element that is the image of the ground term "makeVehicle(2,1)." Furthermore, the relation $\mathcal{R}_{\texttt{Vehicle}}$ only relates vehicles and bicycles with the same number of wheels and passengers. Hence it is easy to show that $\mathcal{R}$ is a simulation relation. For example, $\mathcal{R}$ satisfies the substitution property at the type `Vehicle` because whenever $q\ \mathcal{R}_{\texttt{Vehicle}}\ r$,

$$\texttt{wheels}^C(q) = \texttt{wheels}^C(r) \quad (4.12)$$

$$\texttt{passengers}^C(q) = \texttt{passengers}^C(r) \quad (4.13)$$

and the same relationships hold fo the corresponding specification functions.

So `Bicycle` is a subtype of `Vehicle`.

The paradigm exhibited by this example can be followed by other specifications with virtual types. That is, one specifies a trait for the virtual type that describes objects of the type that cannot be created by programs. These "virtual objects" are used only in the description of the simulation relation when giving a subtype's specification. Here the two-tiered specification method pays a big dividend, by allowing the objects of a virtual type to be described, even though they do not exist in real programs.

4.3 Other Definitions of Subtype

In this section definition of subtype relations given above is compared with other notions of subtyping. The comparison is limited to technical points. A more general comparison is found in Chapter 1. The major points that emerge are that the subtype relations:

- are defined with a level of formality that allows treatment of subtle issues such as subtyping for incomplete specifications, as opposed to various informal definitions
- are behavioral, since they are based on the specifications of abstract types, as opposed to structural rules for subtyping among certain built-in

types (such as those proposed by Cardelli and others) or structural comparisons between the signatures of abstract types (as in Emerald [BHJL86] [BHJ+87]), and

- generalize closely related work (by Reynolds and by Bruce and Wegner) in that they are defined using simulation *relations* instead of coercion *functions* among families of algebraic models of specifications, and can thus handle nondeterministic and incompletely specified types.

The great advantage of a behavioral notion of subtyping is that it allows a subtype to be implemented in a way that is different than its supertypes. In particular a subtype does not have to be implemented as a subclass of the classes that implements its supertypes, neither does a subtype have to be implemented using a representation that is structurally a subtype of its supertypes. The great advantage of nondeterminism and incomplete specifications is that they allow the specifier more freedom.

In contrast to my dissertation [Lea89], observations are not used to define subtype relations. Some of the reasons for this change have already been mentioned. A detailed discussion of the relation of the definition of subtype relations given above to the one in [Lea89] is found in Chapter 7. For now, it is enough to know that the definition of subtype relations found in [Lea89] is weaker than that given here.

Formal work on the question of when one type is a subtype of another can be roughly divided into two camps: algebraic model theory and type theory. The work of Ait-Kaci [AK84] falls outside this classification. However, Ait-Kaci's characterization of inclusion among terms (i.e., objects), does not help one to answer subtyping questions, since Ait-Kaci takes a partial order among head symbols as given.

4.3.1 Informal Definitions of Subtype Relationships

Schaffert *et al.* offer the following informal definition of a subtype relationship: "Given a type `S` which is a subtype of a type `T`, then any object of type `S` behaves like a `T` object and may be used wherever a `T` object may be used" [SCB+86, Section 5]. For types that are not incompletely specified, this definition seems to agree with the above definition of subtyping. However, for incompletely specified types a simple relation on objects does not suffice to prove a subtype relation, as is shown by the types `PSchd` and `Crowd` (discussed above). That is, each `PSchd` object acts like some `Crowd` object, but `PSchd` is not a subtype of `Crowd`.

Snyder [Sny86b, Page 41] offers the following definition of a subtype relationship: "If instances of class x meet the external interface of class y, then x should be a subtype of y." By "external interface" Synder means a behavioral specification; thus his definition of subtype relationships is also a semantic relationship between instances. For example, he says that "behavioral subtyping cannot be deduced without formal semantic specification of behavior." However, he goes on to say that "lacking such specifications, one can deduce subtyping based solely on syntactic external interfaces (i.e., the names of the operations) [Car84]." This latter statement is wrong, since for valid reasoning based on subtype relationships among abstract types, the semantics of a type must be taken into account. For example, the types `IntSet` and `Interval` have the same set of instance operations, but `IntSet` is not a subtype of `Interval`. Although Snyder cites Cardelli's paper [Car84] to support his statement, Cardelli's syntactic deductions do not apply to abstract types in general, but only to a limited set of types.

4.3.2 Algebraic Approaches

The definition of subtype relations given in this report follow the algebraic tradition of Goguen, Reynolds, and others.

Subtypes as Subsets. In Goguen's work, the signature of an order-sorted algebra has a partial order on sorts, called the subsort relation [GM87]. In an order-sorted algebra, if `S` is a subsort of `T`, then the carrier set of `S` must be a subset of the carrier set of `T`.

The idea that a subtype is semantically a subset has been used in an attempt to explain subtyping in programming languages with higher order types by Cardelli and Wegner [CW85, Page 490]. For certain languages the ideal model [MS82] cited by Cardelli and Wegner, has a rich enough structure to accommodate this explanation. But for Cardelli and Wegner's language, the ideal model is not sound [BL88]. One can also construct special-purpose models with the property that the set of values of a subtype is a subset of the set of values of its supertypes [Car84].

However, building specialized models is not a practical approach to settling subtyping questions. The constraint that a subtype's carrier set must be a subset of each of its supertype's makes such models counter-intuitive and unnecessarily difficult to construct. For example, consider the following immutable record types: `oneBool = record[x: Bool]` and `twoBool = record[x: Bool, y: Bool]`. Most people who have not seen the particular constructions involved would say that there are only two values of type `oneBool` and four values of type `twoBool`. But to show that `twoBool` is a subtype of `oneBool` Cardelli constructs a model where the set of values of type `twoBool` is a subset of `oneBool`'s [Car84].

Cardelli's "A Semantics of Multiple Inheritance". A good example of such a construction is found in Cardelli's landmark 1984 paper [Car84]. This paper describes subtype relationships among the built-in types of a small programming language, as well as its type checking and semantics. This paper does not deal with arbitrary, user-defined types, and hence Cardelli is able to give simple syntactic rules that determine when one of these types is a subtype of another type. Cardelli also shows that these rules are sound in the sense that they prevent certain errors in programs written his language. Cardelli writes $\mathtt{S} \leq \mathtt{T}$ when `S` is syntactically a subtype of `T`.

The semantics of Cardelli's language are described using a domain $\mathbf{V}$. This domain is constructed so that, whenever $\mathtt{S} \leq \mathtt{T}$, then the carrier set of `S` is a subset of the carrier set of `T` [Car84, Page 62]. Thus the domain $\mathbf{V}$ supports subtype polymorphism, since the instance operations of the built-in types work on all subsets of their domains, and hence on all subtypes.

One may regard Cardelli's **V** as the only algebra in the semantics of a specification of the built-in types of his language. One shows that Cardelli's $\leq$ is a subtype relation by constructing a simulation relation from **V** to **V**. As is the case with all such models where the carrier of each subtype is a subset of the carrier of the superset, the simulation relation can be trivial: for each type `T`, let $\mathcal{R}_{\mathtt{T}}$ be the identity on the carrier set of `T`. Such a family of identity relations trivially satisfies all the required properties, hence Cardelli's $\leq$ is a subtype relation.

The differences are that Cardelli's **V** is designed for function, immutable record, and immutable oneof types, so he is able to handle these types more directly. But Cardelli's construction is not easily adapted to arbitrary abstract types.

Definitions Based on Homomorphic Functions. The definition of subtype relations given above extends the work of Reynolds as well as Bruce and Wegner in that it allows one to compare type specifications instead of particular models. This is important for incompletely specified types, such as `Crowd`, which do not have a single model that exhibits all of the relevant behavior.

Another generalization from the work of Reynolds and Bruce and Wegner is that subtype relations are described using simulation relations instead of functions. This is a practical benefit in specifications, since one does not have to specify carrier sets so carefully that there are no observably equivalent elements. Technically this means that one's specification of a carrier set does not have to be sufficiently complete [GH78].

The following example shows how subtyping based on homomorphic functions (Bruce and Wegner's definition) fails to show a subtype relationship between two types that are observably equivalent. The example is adapted from a paper by Mitchell [Mit86, Page 266].

Let the types `S1` and `S2` represent multi-sets of integers (i.e., bags of integers). Since neither Reynolds nor Bruce and Wegner deal with specifications, one may choose a particular model. So consider an algebra A, where the elements of the carrier set of `S1` are lists (written in square brackets below) of ordered pairs of the form $\langle element, count\rangle$ and the elements of the carrier set of `S2` are just lists of the elements inserted in the order in which they are inserted. Let `newS1` denote the empty multi-set of type `S1` and let `newS2` denote the empty multi-set of type `S2`. Each type also has instance operations `ins` and `count`. To illustrate the representations, consider the following:

$$\begin{aligned}
&\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS1}^A(), 5), 40) && (4.14)\\
&\quad = \quad \{[\langle 5,1\rangle, \langle 40,1\rangle]\} && (4.15)\\
&\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS2}^A(), 5), 40) && (4.16)\\
&\quad = \quad \{[5, 40]\}. && (4.17)
\end{aligned}$$

The example exploits the differences in the representations. Suppose that $q \neq r$; then for the type `S1`, inserting two q's and an r gives the same result in two different orders:

$$\begin{aligned}
&\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS1}^A(), q), r), q)\\
&\quad = \quad \{[\langle q,2\rangle, \langle r,1\rangle]\}\\
&\quad = \quad \mathtt{ins}^A(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS1}^A(), q), q), r)
\end{aligned}$$

but for the type `S2` the order matters:

$$\begin{aligned}
&\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS2}^A(), q), r), q)\\
&\quad = \quad \{[q, r, q]\}\\
&\quad \neq \quad \{[q, q, r]\}\\
&\quad = \quad \mathtt{ins}^A(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS2}^A(), q), q), r).
\end{aligned}$$

For `S1` to be a subtype of `S2` according to Bruce and Wegner's definition, there would have to be a homomorphic function $c_{(1,2)}$ from `S1` to `S2`. By the substitution property, the following equations hold (again assuming that $q \neq r$):

$$\begin{aligned}
&c_{(1,2)}(\mathtt{newS1}^A)\\
&\quad = \quad \mathtt{newS2}^A\\
&\quad = \quad \{[]\}\\
&c_{(1,2)}(\mathtt{ins}^A(\mathtt{newS1}^A(), q))\\
&\quad = \quad \mathtt{ins}^A(\mathtt{newS2}^A(), q)\\
&\quad = \quad \{[q]\}\\
&c_{(1,2)}(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS1}^A(), q), r))\\
&\quad = \quad \{[q, r]\}\\
&c_{(1,2)}(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS1}^A(), q), r), q))\\
&\quad = \quad \{[q, r, q]\}\\
&c_{(1,2)}(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS1}^A(), q), q))\\
&\quad = \quad \{[q, q]\}\\
&c_{(1,2)}(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS1}^A(), q), q), r))\\
&\quad = \quad \{[q, q, r]\}.
\end{aligned}$$

But the only possible result of both

$$\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS1}^A(), q), r), q)$$

and

$$\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{ins}^A(\mathtt{newS1}^A(), q), r), q)$$

is the list of pairs $[\langle q,2\rangle, \langle r,1\rangle]$. Therefore $c_{(1,2)}$ would have to map $[\langle q,2\rangle, \langle r,1\rangle]$ to both $[q, r, q]$ and $[q, q, r]$. Therefore, $c_{(1,2)}$ cannot be a function.

But the types `S1` and `S2` are observably equivalent; that is, their behaviors cannot be distinguished. So Bruce and Wegner's definition is too strong, as it fails to show a subtype relationship among observably equivalent types.

One might object that I "chose the wrong algebras" for the example above, but that is part of the point; neither Reynolds nor Bruce and Wegner show how to take a specification and give it an algebraic model. The other part of the point is that a definition of subtyping based on homomorphic functions is too restrictive.

Chapter 5

An Applicative Language

Subtype relations are intended to aid program verification. To show how they aid program verification, it is necessary to first define a programming language. The language defined in this chapter will also be used in giving examples of observations in Chapter 7.

The language defined in this chapter is called NOAL, which stands for Nondeterministic Object-oriented Applicative Language. It is an applicative language since it has no notion of assignment or mutation. It is nondeterministic so that the claims that subtyping does not allow surprising behavior to be observed will be strong, as only a language with nondeterminism can make certain observations [Nip86]. It is object-oriented because it has a message passing mechanism.

The language NOAL is a hybrid of Trellis/Owl [SCB+86] and Broy's AMPL [Bro86]. NOAL resembles AMPL in that it is a lambda calculus with explicit facilities for nondeterminism. Like AMPL, NOAL is a first-order language; that is, functions are not objects in NOAL programs. NOAL resembles Trellis/Owl in its type system and message passing mechanism.

Type information aids verification, but programs are given a meaning regardless of whether they are type-safe. The type system of NOAL uses the result sort map (*ResSort*) from a signature and the signature's presumed subtype relation ($\leq$) to do type checking.

There are two kinds of nondeterministic primitives in NOAL. The erratic choice expression, `1 [] 2`, has both 1 and 2 as possible results. Operationally, the erratic choice operator picks one expression at random and evaluates it. Therefore, if evaluation of one expression may not halt, then the entire expression may not halt. An angelic choice expression of the form $E_1 \bigtriangledown E_2$ will always halt if either E_1 or E_2 always halts. Operationally, the angelic choice operator runs both expressions in parallel and returns the first result.

NOAL is only half of an object-oriented language, since there is no mechanism for implementing abstract types. Formally, NOAL programs manipulate the algebras described in Chapter 2. The identifiers in a NOAL program denote objects in the carrier set of an algebra. The message passing (or dynamic binding) mechanism is modeled by the algebra itself (in contrast to [Lea89]). The set of possible results of a message send is determined by consulting the appropriate operation of an algebra.

The semantics of NOAL programs makes certain mild assumptions about algebras. The basic assumption is that the visible types, `Bool`, `Int`, `BoolStream` and `IntStream`, as described in Figures B.1, B.2, B.3, form a reduct of each algebra. Some assumptions about the domain ordering are also stated in the section on domains in Appendix C. These conditions are satisfied by the algebraic models of most specifications written in the specification language of Chapter 3. However, NOAL is not restricted to observing models of such specifications.

The syntax of NOAL, its semantics and then its type system are described below.

5.1 NOAL Syntax

A NOAL program consists of mutually recursive function definitions, followed by a program expression, which is invoked to start the program running.

The syntax of NOAL is presented in Figure 5.1. The nonterminal ⟨type⟩ denotes a type symbol, and ⟨message name⟩ denotes a program operation symbol. The syntax of identifiers and function identifiers is left unspecified. However, identifiers and function identifiers that appear in a program may not be the same as the program operation symbols used in that program. (Function identifiers, such as *funName*, are written with a different font than program operation symbols, `pop`.) The symbol `isDef?` may not be used as a function identifier or a program operation symbol.

The following syntactic sugars are also used. Broy's notation for streams is supported by considering an expression of the form E_1 `&` E_2 to be syntactic sugar for `cons(`E_2`,`E_1`)`. A declaration such as `f,s: Int` is sugar for the declaration list `f: Int, s: Int`. A program operation symbol such as `T` used without a list of arguments is sugar for `T()`. The expressions `true` and `false` are sugar for `true(Bool())` and `false(Bool())` and `1`, `2`, and so on are sugar for `one(Int())`, `add(one(Int()),one(Int()))`, and so on.

The following example program has the stream $\langle i, 4\rangle$ as its only possible result, where i is the value of the program's argument.

```
fun pick(s:IntSet):Int = choose(s);
program (i:Int):IntStream =
        i & pick(ins(null(IntSet),4))
        & empty(IntStream)
```

5.2 NOAL Semantics

The meaning of a program is an observation, which is a mapping from an algebra-environment pair to a set of possible results.

Environments were defined in Chapter 3. They map typed identifiers to the elements of the carrier set of an algebra. Recall that a Σ-environment must obey the

```
⟨program⟩ ::= ⟨program expr⟩
    | ⟨rec fun def⟩ ⟨program⟩

⟨program expr⟩ ::= program ⟨heading⟩ = ⟨expr⟩

⟨rec fun def⟩ ::= fun ⟨fun identifier⟩
          ⟨heading⟩ = ⟨expr⟩ ;

⟨heading⟩ ::= ( ⟨decls⟩ ) : ⟨type⟩

⟨decls⟩ ::= ⟨decl list⟩ | ⟨empty⟩
⟨decl list⟩ ::= ⟨decl⟩ | ⟨decl list⟩ , ⟨decl⟩
⟨decl⟩ ::= ⟨identifier⟩ : ⟨type⟩

⟨empty⟩ ::=

⟨expr⟩ ::= ⟨identifier⟩
    | bottom [ ⟨type⟩ ]
    | ⟨message name⟩ ( ⟨exprs⟩ )
    | ⟨fun identifier⟩ ( ⟨exprs⟩ )
    | ( ⟨function abstract⟩ ) ( ⟨exprs⟩ )
    | if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩ fi
    | ⟨expr⟩ [] ⟨expr⟩
    | ⟨expr⟩ ▽ ⟨expr⟩
    | isDef?  ( ⟨expr⟩ )
    | ( ⟨expr⟩ )

⟨exprs⟩ ::= ⟨expr list⟩ | ⟨empty⟩
⟨expr list⟩ ::= ⟨expr⟩ | ⟨expr list⟩ , ⟨expr⟩

⟨function abstract⟩ ::= fun ( ⟨decls⟩ ) ⟨expr⟩
```

Figure 5.1: Syntax of NOAL.

presumed subtype relation $\leq$ of the signature Σ in the sense that an environment may only map an identifier `x:T` to an object that has a type `S` such that `S` $\leq$ `T`.

The meaning of an expression is called an internal observation. Let Σ be a signature. Let X be a set of typed identifiers. An *internal Σ-observation with free identifiers from X* is a mapping that takes a Σ-algebra A and a Σ-environment $\eta : Y \to |A|$ such that $X \subseteq Y$, and returns a set of possible results from A. A *Σ-observation with free identifiers from X* is an internal Σ-observation with free identifiers from X such that each possible result has a visible type.

Throughout the following fix a signature

$$\Sigma = \left(\begin{array}{c} SORTS, TYPES, V, \leq, \\ SFUNS, POPS, ResSort \end{array} \right) \tag{5.1}$$

and a Σ-algebra A.

5.2.1 Semantics of NOAL Expressions

The meaning of a NOAL expression is described by the function $\mathcal{M}$, which takes an expression with free identifiers and function identifiers from a set X of typed identifiers and returns an internal observation. Hence, when one applies the result of $\mathcal{M}$ to an algebra-environment pair, one obtains a set of possible results.

The following list gives the denotation of each recursion-free NOAL expression in an environment $\eta : X \to |A|$. For convenience, it is assumed that η also maps typed function identifiers to the denotations of recursively-defined NOAL functions. It is assumed that the algebra returns $\perp$ whenever a specification function or program operation is invoked outside its domain.

- The only possible result of an identifier is its value in the environment η.

$$\mathcal{M}[\![\mathtt{x}]\!](A, \eta) \stackrel{\text{def}}{=} \{\eta(\mathtt{x})\} \tag{5.2}$$

- The only possible result of an expression of the form `bottom [` ⟨type⟩ `]` is $\perp$. The type is only included in the expression to make type-checking easier. Thus, for each type `T`:

$$\mathcal{M}[\![\mathtt{bottom[T]}]\!](A, \eta) \stackrel{\text{def}}{=} \{\perp\}. \tag{5.3}$$

- The possible results of a message send are determined by consulting the algebra for each possible argument list.

$$\mathcal{M}[\![\mathtt{g}(\vec{E})]\!](A, \eta) \stackrel{\text{def}}{=} \bigcup_{\vec{q} \in \mathcal{M}[\![\vec{E}]\!](A,\eta)} \mathtt{g}^A(\vec{q}) \tag{5.4}$$

  If $\vec{E}$ is empty, then $\mathcal{M}[\![\vec{E}]\!](A, \eta) = \{\langle\rangle\}$, so the above means

$$\mathcal{M}[\![\mathtt{g()}]\!](A, \eta) \stackrel{\text{def}}{=} \mathtt{g}^A(). \tag{5.5}$$

  The same idea applies to empty argument lists below.

- The possible results of a recursively defined function invocation are determined similarly, except that the meaning of the function identifier is found in the environment η.

$$\mathcal{M}[\![f(\vec{E})]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{E}]\!](A,\eta)} (\eta(f))(\vec{q}) \quad (5.6)$$

- The set of possible results of a combination is the set of possible results of the body of the function abstract in each environment that extends the original by binding a list of possible arguments to the formals.

$$\mathcal{M}[\![\texttt{(fun (}\vec{\texttt{x}}\texttt{ : }\vec{\texttt{S}}\texttt{) }E_0\texttt{)(}\vec{E}\texttt{)}]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{E}]\!](A,\eta)} \mathcal{M}[\![E_0]\!](A,\eta[\vec{q}/\vec{\texttt{x}}]) \quad (5.7)$$

- The set of possible results for an `if` expression depend on the possible results for the first subexpression.

$$\mathcal{M}[\![\texttt{if }E_1\texttt{ then }E_2\texttt{ else }E_3\texttt{ fi}]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{q\in\mathcal{M}[\![E_1]\!](A,\eta)} \begin{cases} \mathcal{M}[\![E_2]\!](A,\eta) & \text{if } q = \mathit{true} \\ \mathcal{M}[\![E_3]\!](A,\eta) & \text{if } q = \mathit{false} \\ \{\bot\} & \text{otherwise} \end{cases} \quad (5.8)$$

- The set of possible results of an erratic choice expression includes those of both subexpressions.

$$\mathcal{M}[\![E_1 \sqcap E_2]\!](A,\eta) \stackrel{\text{def}}{=} \mathcal{M}[\![E_1]\!](A,\eta) \cup \mathcal{M}[\![E_2]\!](A,\eta) \quad (5.9)$$

- The set of possible results of an angelic choice expression is the same as an erratic choice, except that $\bot$ is a possible result only if it is a possible result of both subexpressions.

$$\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](A,\eta) \stackrel{\text{def}}{=} \begin{cases} \mathcal{M}[\![E_1]\!](A,\eta) \cup (\mathcal{M}[\![E_2]\!](A,\eta) \setminus \{\bot\}) & \text{if } \bot \notin \mathcal{M}[\![E_1]\!](A,\eta) \\ (\mathcal{M}[\![E_1]\!](A,\eta) \setminus \{\bot\}) \cup \mathcal{M}[\![E_2]\!](A,\eta) & \text{if } \bot \notin \mathcal{M}[\![E_2]\!](A,\eta) \\ \mathcal{M}[\![E_1]\!](A,\eta) \cup \mathcal{M}[\![E_2]\!](A,\eta) & \text{otherwise} \end{cases} \quad (5.10)$$

  The set $s_1 \setminus s_2$, consists of the elements of s_1 that are not elements of s_2.

- The primitive `isDef?` can be used to see if an expression's possible results are proper.

$$\mathcal{M}[\![\texttt{isDef?(}E\texttt{)}]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{q\in\mathcal{M}[\![E]\!](A,\eta)} \begin{cases} \{\mathit{true}\} & \text{if } q \neq \bot \\ \{\bot\} & \text{otherwise} \end{cases} \quad (5.11)$$

  Since there is no "assignment statement" the only way to bind objects to identifiers is through function calls or the application of a function abstract. The parameter passing mechanism of NOAL is lazy evaluation, so `isDef?` is needed to define strict functions.

5.2.2 Semantics of Recursive Function Definitions

NOAL programs may begin with a system of mutually recursive function definitions. In the body of a recursively defined function, there can be no free identifiers or function identifiers, besides those of the other recursively defined functions and the function's formal arguments.

Since NOAL "functions" can be nondeterministic, for a given algebra a function definition's denotation, written $\mathcal{F}[\![f]\!](A)$ is a mapping that takes a tuple of arguments and returns a set of possible results.

It is a difficult problem to construct the least fixed points of systems of mutually recursive function definitions in NOAL. The construction of least fixed points is described in Appendix C. What follows are merely some informal explanations and examples.

NOAL uses lazy evaluation for evaluating function arguments [Sch86, Page 181]; Broy calls the rule *call-time-choice*. Like call-by-name, call-time-choice uses delayed evaluation, hence functions written in NOAL need not be strict. However, each actual parameter is only evaluated once; hence formal parameters are not themselves sources of nondeterminism. That is, if a formal argument is mentioned twice in the body of a function abstract, the same value will be substituted in each instance. The following program demonstrates the difference between call-time-choice and call-by-name:

```
fun f(x:Int): Int = add(x,x);
program (): Int = f(0 [] 1).
```

In NOAL's call-time-choice semantics the above program has as possible results both 0 and 2; a result of 1 is not possible with call-time-choice, although it would be possible with call-by-name. Another interesting example is the following program:

```
fun pick(x:Int): Int =
    x ▽ pick(add(x,1));
program (): Int = pick(0).
```

This program has as its set of possible results all positive integers; furthermore, it is guaranteed to terminate! However, if the angelic choice (▽) were replaced with an erratic choice ([]) in this program, the program would also have all positive integers as possible results, but in addition it might not terminate.

5.2.3 Semantics of NOAL Programs

A NOAL program consists of a series of mutually recursive function definitions, and a ⟨program expr⟩. The ⟨expr⟩ that is the body of the ⟨program expr⟩ may use only the identifiers declared in the ⟨program expr⟩'s heading and the function identifiers declared in the program.

The notation $\mathcal{M}$ is also used to denote the function that takes a program and returns an observation.

All possible results of a program are guaranteed to have a visible type through the use of the following function:

$$Visible(q) \stackrel{\text{def}}{=} \begin{cases} q & \text{if } q \in \mathtt{T}^A \text{ and } \mathtt{T} \in V \\ \bot & \text{otherwise.} \end{cases} \quad (5.12)$$

The function *Visible* is extended pointwise to sets of possible results.

The meaning of a program is given by the following definition.

Consider the program: $\vec{F}$; `program (`$\vec{\mathtt{x}}$ `:` $\vec{\mathtt{S}}$`):T =` E with a list of recursive function definitions $\vec{F}$ and an expression E whose free identifiers are the set $X = \{\mathtt{x}_i : \mathtt{S}_i\}$. Let $\eta : Y \to A$ be an environment such that $X \subseteq Y$. Let η' be $\eta[\mathcal{F}[\![\vec{F}]\!](A)/\vec{f}]$; that is, η extended by binding fixed points $\mathcal{F}[\![\vec{F}]\!](A)$ to the function identifiers $\vec{f}$ defined in $\vec{F}$. Then the meaning of the above program is the meaning of E in η'; that is:

$$\begin{aligned} &\mathcal{M}[\![\vec{F}\text{; } \mathtt{program\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{):T =}\ E]\!](A, \eta) \\ &\quad \stackrel{\text{def}}{=} \quad Visible(\mathcal{M}[\![E]\!](A, \eta')). \end{aligned} \quad (5.13)$$

5.3 Nominal Types and Type Checking for NOAL

This section describes the nominal types of NOAL expressions and type checking for NOAL programs.

5.3.1 Nominal Types

The nominal type of an expression plays a crucial role in program verification. An expression's nominal type is an upper bound on the types of the objects it can denote. That is, an expression with nominal type `T` can only denote an object whose type is a subtype of `T`. The notion of an expression's nominal type is similar to Reynolds's notion of the minimal type of an expression [Rey80] [Rey85].

The guarantee implicit in the nominal type of an expression can only be realized if the expression has a certain form. Hence not all expressions have a nominal type, only those that type-check. An expression (or program) that has a nominal type is called *type-safe*. The verification method discussed in Chapter 6 only applies to type-safe programs. Type-safe programs also form an interesting class of observations, since a type-safe program can only observe the results of an expression by invoking the instance operations of the expression's nominal type. For example, a type-safe program cannot apply `leastFirst` to an expression of nominal type `IntSet`, since `leastFirst` is not one of the operations defined on `IntSet`. So when observed by a type-safe program, instances of `PSchd` that are bound to identifiers of nominal type `IntSet` behave like instances of `IntSet`.

Following Reynolds [Rey85], type checking for NOAL programs is described by using a signature's result sort mapping, its presumed subtype relation, and the nominal signatures determined by the declarations of recursively defined functions.

The nominal type of an expression is defined recursively. At the base, the nominal type of an identifier is given in its declaration. The nominal type of a function call is given by that function's declaration. To support subtype polymorphism, an actual argument may have a type that is a subtype of the corresponding formal argument type, and thus the actual argument expressions may have nominal types that are subtypes of the corresponding formal argument types. Similarly, the body of a function may have a nominal type that is a subtype of the nominal result type of the function. The nominal type of a message send is determined by the *ResSort* function, applied to the nominal types determined (recursively) for the arguments.

The treatment of type checking is thus similar to Reynolds's [Rey80]. Like Reynolds, the type inference rules below assign a single nominal type, to each type-safe expression. This is in contrast type systems with a rule of subsumption, such as Cardelli's [Car84], where expressions have multiple types. As with Reynolds's system, the nominal type of an `if` expression is the least upper bound of the nominal types of the arms, if the least upper bound exists. In Reynolds's system, there is a type *ns* (nonsense) that is a supertype of all other types, but the NOAL type system has no such type.

5.3.2 Type Checking

Figure 5.2 shows the type inference rules for NOAL. These rules precisely define the *nominal type* of each NOAL expression. In the figure, H is a type environment that maps identifiers to types and function identifiers to nominal signatures. A type environment H can be thought of as a set of type assumptions, which are the pairs of the mapping. An assumption of the form $\mathtt{x} : \mathtt{T}$ means that the identifier `x` has nominal type `T`. An assumption of the form $f : \vec{\mathtt{S}} \to \mathtt{T}$ means that the function identifier f has nominal signature $\vec{\mathtt{S}} \to \mathtt{T}$. The notation $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ means that each $\mathtt{x}_i$ has nominal type $\mathtt{S}_i$. The notation $H, \mathtt{x} : \mathtt{T}$ means $H[\mathtt{T}/\mathtt{x}]$; that is, H extended with the assumption $\mathtt{x} : \mathtt{T}$ (where the extension replaces all assumptions about `x` in H). The notation $\Sigma; H \vdash E : \mathtt{T}$ means that given the signature Σ and the type environment H one can prove that the expression E has nominal type `T` using the inference rules. The notation $\Sigma \vdash \mathrm{lub}(\mathtt{S}, \mathtt{U}) = \mathtt{T}$ means that the least upper bound in Σ's presumed subtype relation, $\leq$, of `S` and `U` exists and is equal to `T`. The notation $\Sigma \vdash \vec{\sigma} \leq \vec{\mathtt{S}}$ means that for each i, $\sigma_i \leq \mathtt{S}_i$, where $\leq$ is the presumed subtype relation of Σ. An inference rule of the form:

$$\frac{h_1, h_2}{c}$$

means that to prove the conclusion c one must first show that both hypotheses h_1 and h_2 hold. Rules written without hypotheses and the horizontal line are axioms.

The only rules that allow one to exploit the presumed subtype relation $\leq$ are [mp], [fcall] and [comb]. These rules allow the nominal type of an actual argument expression to be a presumed subtype of the formal's type.

Let H be a mapping from function identifiers to function signatures. The *set of type-safe NOAL ex-*

[ident] $\Sigma; H, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}$

[fident] $\Sigma; H, f : \vec{\mathtt{S}} \to \mathtt{T} \vdash f : \vec{\mathtt{S}} \to \mathtt{T}$

[mp] $$\frac{\Sigma; H \vdash \vec{E} : \vec{\sigma},\ \mathit{ResSort}(\mathtt{g}, \vec{\sigma}) = \mathtt{T}}{\Sigma; H \vdash \mathtt{g(}\vec{E}\mathtt{)} : \mathtt{T}}$$

[fcall] $$\frac{\Sigma; H \vdash f : \vec{\mathtt{S}} \to \mathtt{T},\ \Sigma; H \vdash \vec{E} : \vec{\sigma},\ \Sigma \vdash \vec{\sigma} \leq \vec{\mathtt{S}}}{\Sigma; H \vdash \mathtt{f(}\vec{E}\mathtt{)} : \mathtt{T}}$$

[comb] $$\frac{H, \vec{\mathtt{x}} : \vec{\mathtt{S}} \vdash E_0 : \mathtt{T},\ \Sigma; H \vdash \vec{E} : \vec{\sigma},\ \Sigma \vdash \vec{\sigma} \leq \vec{\mathtt{S}}}{\Sigma; H \vdash \mathtt{(fun\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{)}\ E_0\mathtt{)\ (}\vec{E}\mathtt{)} : \mathtt{T}}$$

[if] $$\frac{\begin{array}{c}\Sigma; H \vdash E_1 : \mathtt{Bool},\\ \Sigma; H \vdash E_2 : \mathtt{S}_2,\ \Sigma; H \vdash E_3 : \mathtt{S}_3,\\ \Sigma \vdash \mathrm{lub}(\mathtt{S}_2, \mathtt{S}_3) = \mathtt{T}\end{array}}{\Sigma; H \vdash \mathtt{(if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{fi)} : \mathtt{T}}$$

[erratic] $$\frac{\begin{array}{c}\Sigma; H \vdash E_1 : \mathtt{S}_1,\ \Sigma; H \vdash E_2 : \mathtt{S}_2,\\ \Sigma \vdash \mathrm{lub}(\mathtt{S}_1, \mathtt{S}_2) = \mathtt{T}\end{array}}{\Sigma; H \vdash (E_1 \sqcap E_2) : \mathtt{T}}$$

[angelic] $$\frac{\begin{array}{c}\Sigma; H \vdash E_1 : \mathtt{S}_1,\ \Sigma; H \vdash E_2 : \mathtt{S}_2,\\ \Sigma \vdash \mathrm{lub}(\mathtt{S}_1, \mathtt{S}_2) = \mathtt{T}\end{array}}{\Sigma; H \vdash (E_1 \bigtriangledown E_2) : \mathtt{T}}$$

[isDef] $$\frac{\Sigma; H \vdash E : \mathtt{S}}{\Sigma; H \vdash \mathtt{isDef?(}E\mathtt{)} : \mathtt{Bool}}$$

[bot] $\Sigma; H \vdash \mathtt{bottom[T]} : \mathtt{T}$

[prog] $$\frac{\begin{array}{c}\Sigma; f_1 : \vec{\mathtt{S}_1} \to \mathtt{T}_1, \ldots, f_m : \vec{\mathtt{S}_m} \to \mathtt{T}_m, \vec{\mathtt{x}_1} : \vec{\mathtt{S}_1} \vdash E_1 : \mathtt{T}_1,\\ \vdots\\ \Sigma; f_1 : \vec{\mathtt{S}_1} \to \mathtt{T}_1, \ldots, f_m : \vec{\mathtt{S}_m} \to \mathtt{T}_m, \vec{\mathtt{x}_m} : \vec{\mathtt{S}_m} \vdash E_m : \mathtt{T}_m,\\ \Sigma; \vec{\mathtt{y}} : \vec{\mathtt{U}}, f_1 : \vec{\mathtt{S}_1} \to \mathtt{T}_1, \ldots, f_m : \vec{\mathtt{S}_m} \to \mathtt{T}_m \vdash E : \mathtt{T},\\ \mathtt{T} \in V\end{array}}{\Sigma \vdash \left(\begin{array}{l}\mathtt{fun}\ f_1\ \mathtt{(}\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}\mathtt{):T}_1\ \mathtt{=}\ E_1\mathtt{;}\\ \vdots\\ \mathtt{fun}\ f_m\ \mathtt{(}\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}\mathtt{):T}_m\ \mathtt{=}\ E_1\mathtt{;}\\ \mathtt{program\ (}\vec{\mathtt{y}} : \vec{\mathtt{U}}\mathtt{):T}\ \mathtt{=}\ E\end{array}\right) : \mathtt{T}}$$

Figure 5.2: Type Inference Rules for NOAL

pressions over Σ *and* H is the set of all NOAL expressions that have a nominal type, when type-checked against the signature Σ and H. The *set of type-safe NOAL programs over* Σ is the set of all NOAL programs whose nominal type is a visible type. If *SPEC* is a specification, then the phrase "the set of type-safe programs over *SPEC*" means the set of type-safe NOAL programs over the signature $SIG(SPEC)$.

How is the nominal type of an expression affected by adding new types to a program? This question is important for modularity of program verification, since type checking is part of the verification process. Therefore modular type checking is also important to modular verification. Adding new types may change the nominal types of expression, but if the new types are added in such a way as to make the original signature a subsignature of the new signature, the new nominal types will only be subtypes of the original nominal types.

The following lemma is thus the source of some of the restrictions in the definition of subsignature. In particular, when one adds new types, one cannot relate previously unrelated types by the presumed subtype relation. Furthermore, since least upper bounds (in $\leq$) are used in several places during type checking, and if two original types `S` and `T` had a least upper bound `U`, then `S` and `T` must have a least upper bound that is a presumed subtype of `U` in the expanded signature; also the subtypes of `S` and `T` must have least upper bounds that are subtypes of `U`. This last property ensures that adding new types does not cause the least upper bounds of certain expressions to become larger than expected in the original signature or undefined.

Lemma 5.3.1. Let Σ' and Σ be signatures. Let $\leq$ be the presumed subtype relation of Σ. Let H be a type environment. Let $SORTS'$ and $SORTS$ be the sort symbols of Σ' and Σ. Let $\mathtt{T} \in SORTS'$ be a sort symbol. Let E be a NOAL expression.

If Σ' is a subsignature of Σ and $\Sigma'; H \vdash E : \mathtt{T}$, then there is some type $\mathtt{S} \in SORTS$ such that $\mathtt{S} \leq \mathtt{T}$ and $\Sigma; H \vdash E : \mathtt{S}$.

Proof: (by induction on the length of the proof of $\Sigma'; H \vdash E : \mathtt{T}$).

Suppose that $\Sigma'; H \vdash E : \mathtt{T}$.

For the basis, if the proof has one step, then it must consist of an instance of one of the axiom schemes [ident] or [bot]. If E is an identifier `x`, then its nominal type is determined by H, so $\Sigma; H \vdash \mathtt{x} : \mathtt{T}$. If E is `bottom[T]`, then $\Sigma; H \vdash \mathtt{bottom[T]} : \mathtt{T}$ is an axiom. Since $\leq$ is reflexive, $\mathtt{T} \leq \mathtt{T}$.

Suppose that the proof of $\Sigma'; H \vdash E : \mathtt{T}$ takes $n > 1$ steps. The inductive hypothesis is that if $\Sigma'; H \vdash E_1 : \mathtt{T}_1$ is any step of the proof but the last, then there is some type $\mathtt{S}_1 \leq \mathtt{T}_1$ such that $\Sigma; H \vdash E_1 : \mathtt{S}_1$. If the last step of the proof is an instance of the axioms [ident], or [bot], then the result follows as above. Otherwise there are several cases.

- If the last step of the proof is an the conclusion of the rule [mp], then E has the form $\mathtt{g}(\vec{E})$. There must be earlier steps of the form $\Sigma'; H \vdash \vec{E} : \vec{\tau}$ and $\Sigma' \vdash \mathit{ResSort}'(\mathtt{g}, \vec{\tau}) = \mathtt{T}$. By the inductive hypothesis, $\Sigma; H \vdash \vec{E} : \vec{\sigma}$, where $\vec{\sigma} \leq \vec{\tau}$. Since Σ' is a subsignature of Σ, $\mathit{ResSort}(\mathtt{g}, \vec{\tau}) = \mathtt{T}$. By the monotonicity of *ResSort*, it follows that for some $\mathtt{S} \leq \mathtt{T}$, $\Sigma \vdash \mathit{ResSort}(\mathtt{g}, \vec{\sigma}) = \mathtt{S}$.
- If the last step of the proof is an instance of [fcall] there must be earlier steps in the proof of the form $\Sigma'; H \vdash f : \vec{\mathtt{S}} \to \mathtt{T}$, $\Sigma'; H \vdash \vec{E} : \vec{\sigma}$, and $\Sigma' \vdash \vec{\sigma} \leq' \vec{\mathtt{S}}$. Since the nominal type of f only depends on H, $\Sigma; H \vdash f : \vec{\mathtt{S}} \to \mathtt{T}$. By the inductive hypothesis, there is some $\vec{\tau} \leq \vec{\sigma}$ such that $\Sigma; H \vdash \vec{E} : \vec{\tau}$. Since Σ' is a subsignature of Σ, $\vec{\sigma} \leq \vec{\mathtt{S}}$. Since $\leq$ is transitive, $\vec{\tau} \leq \vec{\mathtt{S}}$.
- If the last step of the proof is an instance of [comb] there must be earlier steps in the proof of the form $\Sigma'; H, \vec{\mathtt{x}} : \vec{\mathtt{S}} \vdash E_0 : \mathtt{T}$, $\Sigma'; H \vdash \vec{E} : \vec{\sigma}$, and $\Sigma' \vdash \vec{\sigma} \leq' \vec{\mathtt{S}}$. By the inductive hypothesis, there is some type $\mathtt{U} \leq \mathtt{T}$ such that $\Sigma; H, \vec{\mathtt{x}} : \vec{\mathtt{S}} \vdash E_0 : \mathtt{U}$. By the inductive hypothesis, there is some $\vec{\tau} \leq \vec{\sigma}$ such that $\Sigma; H \vdash \vec{E} : \vec{\tau}$. Since Σ' is a subsignature of Σ, $\vec{\sigma} \leq \vec{\mathtt{S}}$. Since $\leq$ is transitive, $\vec{\tau} \leq \vec{\mathtt{S}}$.
- If the last step of the proof is an instance of [if] there must be earlier steps in the proof of the form: $\Sigma'; H \vdash E_1 : \mathtt{Bool}$, $\Sigma'; H \vdash E_2 : \mathtt{S}'_2$, $\Sigma'; H \vdash E_3 : \mathtt{S}'_3$, and $\Sigma' \vdash \mathrm{lub}(\mathtt{S}'_2, \mathtt{S}'_3) = \mathtt{T}$. Since there can be no subtypes of `Bool`, by the inductive hypothesis, $\Sigma; H \vdash E_1 : \mathtt{Bool}$. By the inductive hypothesis there are types $\mathtt{S}_2 \leq \mathtt{S}'_2$ and $\mathtt{S}_3 \leq \mathtt{S}'_3$ such that $\Sigma; H \vdash E_2 : \mathtt{S}_2$ and $\Sigma; H \vdash E_3 : \mathtt{S}_3$. Since Σ' is a subsignature of Σ, there is a sort $\mathtt{U} \leq \mathtt{T}$ that is a least upper bound for $\mathtt{S}_2$ and $\mathtt{S}_3$.
- If the last step of the proof is an instance of [erratic] or [angelic], then the result follows as for [if].
- If E is `isDef?(`E'`)`, then the result follows directly from the inductive hypothesis.

∎

5.3.3 Obedience

Type checking in NOAL aids program verification because it ensures that the possible results that an expression may denote all have a subtype of the expression's nominal type. A language with this property is said to *obey* the presumed subtype relation $\leq$ that is used in type checking. The obedience of NOAL is shown formally below by first showing obedience for expressions(in two steps), then for recursively-defined NOAL functions, and finally for programs.

That the evaluation of NOAL expressions is obedient is shown by first showing obedience for expressions that do not involve function calls, and then for expressions with function calls.

The proof of the obedience of expressions without function calls can be regarded as the source of some of the restrictions on signatures (compare with [Rey80]). The condition that $\leq$ be transitive comes from function calls, where the nominal type of a formal may be `S`, the nominal type of the actual may be $\sigma \leq \mathtt{S}$, and the type of the actual argument may be $\sigma' \leq \sigma$. The condition that *ResSort* be monotonic comes from message sends.

Lemma 5.3.2. Let Σ be a signature with result sort map *ResSort* and presumed subtype relation $\leq$. Let A be a Σ-algebra. Let E be a NOAL expression of nominal type `T` whose set of free identifiers is X. Let $\eta : X \to |A|$ be a Σ-environment.

If there are no calls on NOAL functions in E, then each possible result of $\mathcal{M}[\![E]\!](A, \eta)$ has a type $\tau \leq$ `T`.

Proof: (by induction on the structure of expressions.)

The basis for the induction consists of identifiers and the expression `bottom[T]`. If the expression is an identifier `x` : `T`, then $\mathcal{M}[\![\mathtt{x}]\!](A, \eta) = \{\eta(\mathtt{x})\}$ and by hypothesis, the type of $\eta(\mathtt{x})$ is related by $\leq$ to `T`. The result is trivial for `bottom[T]`, since the only possible result is $\perp$.

For the inductive step, assume that the result holds for each subexpression.

- Suppose the expression is a message send `g(`$\vec{E}$`)`. By the type inference rule [mp], $\vec{E} : \vec{\sigma}$ and $\mathit{ResSort}(\mathtt{g}, \vec{\sigma}) = \mathtt{T}$. Let $\vec{q}$ be a tuple of possible results from $\vec{E}$. By the inductive hypothesis, $\vec{q}$ has a type $\vec{\sigma'}$ such that $\vec{\sigma'} \leq \vec{\sigma}$. By the monotonicity of Σ, $\mathit{ResSort}(\mathtt{g}, \vec{\sigma'}) \leq \mathtt{T}$. By the requirements on algebras, each possible result must have a type `S` such that `S` $\leq \mathit{ResSort}(\mathtt{g}, \vec{\sigma'})$. So by transitivity of $\leq$, each possible result's type `S` is such that `S` $\leq$ `T`.
- Suppose the expression is a combination `(fun (`$\vec{\mathtt{x}} : \vec{\mathtt{S}}$`)` E_0`) (`$\vec{E}$`)`. By the type inference rule [comb], E_0 has nominal type `T`, $\vec{E} : \vec{\sigma}$, and $\vec{\sigma} \leq \vec{\mathtt{S}}$. Let $\vec{q}$ be a tuple of possible results from $\vec{E}$. By the inductive hypothesis, $\vec{q}$ has a type $\vec{\sigma'}$ such that $\vec{\sigma'} \leq \vec{\sigma}$. Since $\leq$ is transitive, $\vec{\sigma'} \leq \vec{\mathtt{S}}$. So the environment $\eta[\vec{q}/\vec{\mathtt{x}}]$ obeys $\leq$ and thus the result follows from the inductive hypothesis applied to E_0.
- Suppose the expression is `if` E_1 `then` E_2 `else` E_3 `fi`. Then by the type inference rule [if], E_1 has nominal type `Bool`. Since there are no subtypes of `Bool`, by the inductive hypothesis, it follows that all possible results of E_1 have type `Bool`. The possible results of the entire expression, therefore, are either $\perp$ or results of E_2 or E_3. By the inductive hypothesis, each possible result of E_2 has a type $\mathtt{S}'_1 \leq \mathtt{S}_1$. Since $\mathtt{S}_1 \leq \mathtt{T}$, the type of each result of E_2 is a subtype of `T`. Similarly for the results of E_3.
- The case for erratic choice and angelic choice expressions is similar to that for [if].
- The `isDef?` expression can only return either $\perp$ or *true*.

∎

The following lemma considers the general case of an expression that may call a recursively defined NOAL function. The proof considers each possible result and shows a type-safe computation that produces it that does not involve function calls.

Lemma 5.3.3. Let Σ be a signature with result sort map *ResSort* and presumed subtype relation $\leq$. Let A be a Σ-algebra. Let $\vec{F}$ be a type-safe system of mutually recursive NOAL function definitions. Let E be a NOAL expression of nominal type `T` whose set of free identifiers is X. Let $\eta : X \to |A|$ be a Σ-environment. Let η' be $\eta[\vec{f}/\vec{f}]$; that is, η extended by binding fixed points $\vec{f}$ to the function identifiers $\vec{f}$ defined in $\vec{F}$.

Then each possible result of $\mathcal{M}[\![E]\!](A, \eta')$ has a type $\tau \leq$ `T`.

Proof: Let $q \in \mathcal{M}[\![E]\!](A, \eta')$ be a possible result. If $q = \perp$ the lemma holds, so suppose $q \neq \perp$. Pick a computation that produces q. Since $q \neq \perp$, the computation uses only finitely many calls, say n, to the f_i.

Expand E by replacing each call to a function $f_i \in \vec{F}$ of the form f_i`(`$\vec{E}$`)` with a combination of the form

`(fun (`$\vec{\mathtt{x}} : \vec{\mathtt{S}}$`)` E_i`) (`$\vec{E}$`)`

where the $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ are the formal arguments and E_i is the body of f_i. Repeat this process on the resulting expression n times and then replace all the remaining function calls with the expression `bottom[`S`]`, where S is the nominal result type of the replaced call.

The above expansion does not change the nominal type of the resulting expression. By construction the set of possible results of the expansion includes q. There are no free function identifiers that remain. So the previous lemma applies. ∎

Programs as a whole also obey the presumed subtype relation, as shown in the following lemma.

Lemma 5.3.4. Let Σ be a signature with result sort map *ResSort* and presumed subtype relation $\leq$. Let P be a NOAL program of nominal type `T` whose formal arguments are $\vec{\mathtt{x}} : \vec{\mathtt{S}}$. Let $X = \{\mathtt{x}_i : \mathtt{S}_i\}$ and let $\eta : Y \to A$ be a Σ-environment such that $X \subseteq Y$.

Then each possible result of $\mathcal{M}[\![P]\!](A, \eta)$ has a type $\tau \leq$ `T`.

Proof: In general P has the form $\vec{F}$`; program (`$\vec{\mathtt{x}}$: $\vec{\mathtt{S}}$`):T =` E. By definition, the possible results of P are given by

$$\mathcal{M}[\![P]\!](A, \eta) = \mathit{Visible}(\mathcal{M}[\![E]\!](A, \eta')),$$

where η' is $\eta[\vec{f}/\vec{f}]$; that is, η extended by binding fixed points $\vec{f}$ to the function identifiers $\vec{f}$ defined in $\vec{F}$.

Since P is type-safe, so are the recursively defined functions. Hence the previous lemma applies. Since `T` is a visible type, *Visible* does not change the set of possible results. ∎

Chapter 6

Hoare-style Verification for NOAL Programs

A Hoare logic for the modular verification of NOAL programs is presented in this chapter and shown to be sound. The logic itself is straight-forward, because of the restrictions on signatures that ensure that assertions that apply to objects of a type `T` also apply to objects that are subtypes of `T`. The key to the soundness proof is the existence of a simulation relation which is guaranteed if the specified relation on types, $\leq$, is truly a subtype relation. A discussion of modularity and some conclusions about how the NOAL type system aids verification follow the presentation of the logic and the soundness results.

During verification one essentially ignores subtyping. This allows a separation of concerns: one specifies types and proves the specified relation on types $\leq$ is a subtype relation, then during verification one can ignore subtyping. Because of this separation of concerns, if one verifies a NOAL function and then later adds new subtypes to some of the function's nominal argument types, then the verification does not have to be repeated.

As usual, the specifications of each type's operations and the specification of each recursively-defined NOAL function is taken as an axiom. The axiom used for a particular message send is determined by the nominal types of the message send's arguments. Note that in verification the overloading is *static*. The point of the definition of subtype relations is to ensure that the static overloading done during verification bears a proper relationship to the dynamic overloading caused by message passing during program execution.

A verification technique is *sound* if whenever one concludes by using that technique that a program satisfies its specification then that program does indeed satisfy its specification. The soundness proof follows the presentation of the logic below.

6.1 A Hoare Logic for NOAL

The Hoare logic [Hoa69] of NOAL programs is a total correctness logic. That is, specifications require termination whenever their pre-condition is met. The logic is sound, but it is not complete, since there is no method given for reasoning about nontermination as would be required to deal with NOAL's lazy evaluation and angelic choice expressions in a complete way.

Although NOAL is applicative, a Hoare logic is used instead of equational reasoning because NOAL is nondeterministic and because the ultimate goal of this research is the verification of imperative programs, for which Hoare-style reasoning is an accepted technique.

The main formulas of a Hoare logic are called Hoare-triples. Hoare-triples are written $P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$ or

$$\begin{array}{l} P \\ \{\mathtt{v} : \mathtt{T} \leftarrow E\} \\ Q \end{array}$$

and consist of a *pre-condition* P, a *result identifier* `v` : `T`, an expression E, and a *post-condition* Q. In a Hoare logic for an imperative language, the pre-condition describes the state before the execution of a statement, and the post-condition describes the changed state that results from the statement's execution. In NOAL, however, expressions have results but do not change the environment in which they execute. Instead the pre-condition of a Hoare-triple describes the environment, and the post-condition describes the environment that results from binding the result identifier (`v`) to a possible result of the expression. So that the notation does not cause confusion, the result identifier in a Hoare-triple cannot occur free in the pre-condition. Otherwise one might think that the execution of E changes the binding of the result identifier in the surrounding environment, whereas the notation only shows what identifier will be used to denote the possible results of E in the post-condition.

In the verification of NOAL programs one is concerned with both sets of type specifications and function specifications. So in this chapter specifications will often consist of pairs $(SPEC, FSPEC)$, where $SPEC$ is a set of type specifications, and $FSPEC$ is a set of function specifications whose base specification set is included in $SPEC$. For a set $FSPEC$ of NOAL function specifications, $SIG(FSPEC)$ consists of a mapping from function identifiers to nominal signatures.

Definition 6.1.1 (Hoare-triple).
Let $(SPEC, FSPEC)$ be a pair of type and function specification sets. Let `T` be a type symbol from of $SIG(SPEC)$. Let $\leq$ be the presumed subtype relation of $SIG(SPEC)$. Then the formula $P\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ Q$ is a *Hoare-triple for* $(SPEC, FSPEC)$ if and only if there is some type $\mathtt{S} \leq \mathtt{T}$ such that given $SIG(SPEC)$ and $SIG(FSPEC)$ one can prove that E has nominal type `S`, P and Q are $SIG(SPEC)$-assertions, and `y` does not occur free in P.

The pair $(SPEC, FSPEC)$ is omitted when clear from context. The type of the result identifier is also usually omitted.

The nominal type of the result identifier can be a supertype of the nominal type of E, because assertions

can describe objects of subtypes of the nominal types of their free identifiers.

Intuitively, $P\ \{\mathtt{v} \leftarrow E\}\ Q$ is true if whenever P holds, then the execution of E terminates, and all possible results satisfy Q. The semantics of a Hoare-triple is given by the following definition, which is similar to the definition of satisfies for function specifications.

Definition 6.1.2 (models).
Let $(SPEC, FSPEC)$ be a pair of type and function specification sets. Let $P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$ be a Hoare-triple for $(SPEC, FSPEC)$. Let X be a set of free identifiers that contains all the free identifiers of P, E, and Q except $\mathtt{v} : \mathtt{T}$. Let A be a $SPEC$-algebra, and let $\eta : X \to |A|$ be a proper $SIG(SPEC)$-environment. For each f in the domain of $SIG(FSPEC)$, let $\mathcal{F}[\![f]\!]$ be a function denotation with signature $SIG(FSPEC)(f)$. Let η' be the environment that extends η by binding each free function identifier f in the domain of $SIG(FSPEC)$ to $\mathcal{F}[\![f]\!](A)$. Then (A, η') *models* $P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$, written

$$(A, \eta') \models P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q,$$

if and only if whenever $(A, \eta) \models P$, then for all possible results $r \in \mathcal{M}[\![E]\!](A, \eta')$:

$$r \neq \bot \tag{6.1}$$
$$(A, \eta[r/\mathtt{v}]) \models Q, \tag{6.2}$$

and there is some type $\mathtt{S}$ such that $r \in \mathtt{S}^A$ and $\mathtt{S} \leq \mathtt{T}$.

For example, the Hoare-triple

$$P\ \{\mathtt{v} \leftarrow E\}\ \text{true}$$

means that evaluation of E in environments that model P always terminates, since the post-condition "true" is modeled by every algebra-environment pair. If the pre-condition P is not logically equivalent to "false," then there are no expressions E such that

$$P\ \{\mathtt{v} \leftarrow E\}\ \text{false}$$

is valid, because no algebra-environment pair models the assertion "false." However, for all expressions E and all assertions Q, every algebra-environment pair models the Hoare-triple

$$\text{false}\ \{\mathtt{v} \leftarrow E\}\ Q,$$

since the pre-condition "false" cannot be satisfied.

The Hoare-triple $P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$ is *valid for* $(SPEC, FSPEC)$, written

$$(SPEC, FSPEC) \models P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q,$$

if and only if for all $SPEC$-algebras A for all proper $SIG(SPEC)$-environments $\eta : X \to |A|$ such that X contains the free identifiers of P and E and Q, and for all extensions η' of η that bind each free function identifier f of E to $\mathcal{F}[\![f]\!](A)$, where $\mathcal{F}[\![f]\!]$ is a denotation that satisfies the specification of f in $FSPEC$ with respect to $SPEC$, $(A, \eta') \models P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$.

Figures 6.1 and 6.2 contain the proof rules for NOAL expressions. In these figures, P, Q, and R are assertions, M is a term, and E, E_1, and so on are NOAL expressions. The notation $(SPEC, FSPEC) \vdash H$, where H is a Hoare-triple, means that one can prove H using the proof rules, including the traits and specifications of $(SPEC, FSPEC)$. The notation $\vdash H$ will be used when the specification pair $(SPEC, FSPEC)$ is clear from context. The notation $SPEC \vdash Q$ means that the formula Q is provable from the traits of $SPEC$. A proof rule of the form:

$$\frac{h_1, h_2}{c}$$

means that to prove the conclusion c one must first show that both hypotheses h_1 and h_2 hold. Rules written without hypotheses and the horizontal line are axiom schemes.

Each rule is named, for convenience in proofs. The name of a rule appears to the left of that rule. To the right of some of the rules are conditions on types and identifiers. Some of the conditions require an identifier to be *fresh*, which means it is not in the set of free identifiers of either the desired pre-condition or the desired post-condition. The conditions of some rules require assertions to be subtype-constraining, abbreviated by "sub.-con." Recall that an assertion is subtype-constraining if the only use of "=" is between terms of visible sort. For example, $\mathtt{x} = 1$ is subtype-constraining, but $\mathtt{y} = \{\}$ is not subtype-constraining.

The rules in Figures 6.1 and 6.2 are discussed below, including the conditions that accompany each rule. For purposes of this discussion, fix a specification $(SPEC, FSPEC)$, which determines a presumed subtype relation $\leq$, a result sort map $ResSort$, and the nominal signatures of NOAL function identifiers. All assertions mentioned are $SIG(SPEC)$-assertions.

- The rule [ident] is an axiom scheme for all types $\mathtt{T}$, for all identifiers $\mathtt{x}$ and $\mathtt{v}$ of nominal type $\mathtt{T}$. The rule says that the only possible result of an expression $\mathtt{x}$ is the value of $\mathtt{x}$.

- The rule [bot] says that `bottom[T]` never terminates.

- The rule [mp-a] is an axiom scheme for all program operation specifications of $SPEC$. The notation $\text{Pre}(\mathtt{g}, \vec{\mathtt{S}})$ means the pre-condition of the operation specification named $\mathtt{g}$ with nominal signature $\vec{\mathtt{S}} \to \mathtt{T}$, where $\mathtt{T} = ResSort(\mathtt{g}, \vec{\mathtt{S}})$. A specification must have at most one such operation specification, since otherwise it would not be legal. Similarly, $\text{Post}(\mathtt{g}, \vec{\mathtt{S}})$ is the post-condition of the operation specification with nominal signature $\vec{\mathtt{S}} \to \mathtt{T}$.

  Both $\text{Pre}(\mathtt{g}, \vec{\mathtt{S}})$ and $\text{Post}(\mathtt{g}, \vec{\mathtt{S}})$ must be subtype-constraining, as required by the specification language of Chapter 3. Technical justification for these restrictions is found below in the soundness proof.

  The axiom only describes the effect of a message send where the actual argument expressions and the result identifier, as well as their types, are

[ident] $(SPEC, FSPEC) \vdash \text{true}\ \{\mathtt{v} : \mathtt{T} \leftarrow \mathtt{x}\}\ \mathtt{v} = \mathtt{x}$ $\qquad \mathtt{x} : \mathtt{T}$

[bot] $(SPEC, FSPEC) \vdash \text{false}\ \{\mathtt{v} : \mathtt{T} \leftarrow \mathtt{bottom[T]}\}\ \text{true}$

[mp-a] $(SPEC, FSPEC) \vdash \text{Pre}(\mathtt{g}, \vec{\mathtt{S}})\ \{\mathtt{y} : \mathtt{T} \leftarrow \mathtt{g}(\vec{\mathtt{x}})\}\ \text{Post}(\mathtt{g}, \vec{\mathtt{S}})$ $\qquad \text{Formals}(\mathtt{g}, \vec{\mathtt{S}}) = \vec{\mathtt{x}} : \vec{\mathtt{S}}, \mathtt{y} : \mathtt{T}$; $\text{Pre}(\mathtt{g}, \vec{\mathtt{S}})$ sub.-con.; $\text{Post}(\mathtt{g}, \vec{\mathtt{S}})$ sub.-con.

[fcall-a] $(SPEC, FSPEC) \vdash \text{Pre}(f, \vec{\mathtt{S}})\ \{\mathtt{y} : \mathtt{T} \leftarrow f(\vec{\mathtt{x}})\}\ \text{Post}(f, \vec{\mathtt{S}})$ $\qquad \text{Formals}(f, \vec{\mathtt{S}}) = \vec{\mathtt{x}} : \vec{\mathtt{S}}, \mathtt{y} : \mathtt{T}$; $\text{Pre}(f, \vec{\mathtt{S}})$ sub.-con.; $\text{Post}(f, \vec{\mathtt{S}})$ sub.-con.

Figure 6.1: Axiom Schemes for verification of NOAL Expressions.

exactly the same as the formal arguments and the formal result used in the specification of $\mathtt{g}$ with nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$. That is, the meaning of $\text{Formals}(\mathtt{g}, \vec{\mathtt{S}}) = \vec{\mathtt{x}} : \vec{\mathtt{S}}, \mathtt{y} : \mathtt{T}$ is that the formal arguments of the relevant specification of $\mathtt{g}$ are $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ and the formal result is $\mathtt{y} : \mathtt{T}$.

- The rule [fcall-a] is an axiom scheme for all function specifications in *FSPEC*. The pre-condition and post-condition of a function come from its specification and must be subtype-constraining, as required by the specification language. Technical justification for this restriction is found in the section on modularity below. The same notation is used as in the rule [mp-a] to denote the pre-condition, post-condition, and formal arguments from the specification of a function. However, the pre-condition and post-condition of a function specification do not depend on the types of the arguments, as there is only one function specified with a given function identifier.

- The inference rule [mp-b] handles the general form of a message passing expression. To prove a triple involving a message send one is obliged to first rewrite the message send from its general form into one where the actual argument expressions are first bound to identifiers. The types of these identifiers should be chosen so that an instance of the rule [mp-a] will apply. The types of the formal arguments of the combination are constrained so that the rewritten expression type-checks.

  Putting the above constraints together, one would normally rewrite a message send to a combination where the nominal type of each actual is a subtype of the nominal type of the corresponding formal and such that there is a program operation specification with the chosen formal argument types. That such a specification always exists is guaranteed by the monotonicity of the *ResSort* map (i.e., by the conditions on signatures). To avoid loss of information, one would normally want the least such program operation specification such that the nominal types of the formals were supertypes of the nominal types of the corresponding actual argument expressions. One could also reason from a less specific operation specification, as this is also permitted by the rule.

- The inference rule [fcall-b] is like [mp-b] in that it requires one to rewrite a general function call so that the argument expressions are first bound to identifiers. These identifiers should be chosen to match the formal arguments from the function's specification.

- The inference rule [comb] handles combinations that may include explicit use of subtyping. The rule as a whole says that to prove that the desired triple holds, one first chooses conjuncts R_i that are sufficient to prove the desired post-condition from the body of the combination.

  The notation $(R_i[\mathtt{v}_i/\mathtt{x}_i])[\vec{\mathtt{x}}/\vec{\mathtt{z}}]$ means the formula R_i with $\mathtt{v}_i$ replacing $\mathtt{x}_i$ throughout and then each $\mathtt{x}_i$ is simultaneously substituted for $\mathtt{z}_i$. The fresh identifiers $\vec{\mathtt{z}}$ are used to hide bindings of $\vec{\mathtt{x}}$ in the assertions that characterize the arguments to the function abstract, so that in reasoning about E_0 the proper scope applies. That is, bindings of the $\mathtt{x}_i$ in the desired pre-condition or the desired post-condition do not mean the $\mathtt{x}_i$ that are local to the body of the function abstract. When reasoning about E_0, the $\mathtt{z}_i$ denote the values of the $\mathtt{x}_i$ in the outer scope. Thus the nominal types of the $\mathtt{z}_i$ should match the nominal types of the $\mathtt{x}_i$ in the outer scope, not the nominal types of the formal arguments to the combination.

  The assertions R_i, may contain the formal argument identifiers, $\mathtt{x}_i$, and thus may be written using the specification functions that apply at the types $\mathtt{S}_i$. The assertions $R_i[\mathtt{v}_i/\mathtt{x}_i]$ will sort-check (and be meaningful) because the nominal type of $\mathtt{v}_i$ is the nominal type of E_i, which must be a presumed subtype of $\mathtt{S}_i$.

  The identifiers $\mathtt{z}_i$ must be fresh to avoid capture problems. The result identifier $\mathtt{y}$ must not be one of the $\mathtt{x}_i$ to avoid capture problems.

- The inference rule [if] allows one to reason about `if` expressions whose boolean expression (E_1) may be nondeterministic. Two special cases are considered before explaining the rule in general.

[mp-b] $$\frac{(SPEC, FSPEC) \vdash P \ \left\{\mathtt{y} \leftarrow \mathtt{(fun\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{)\ g(}\vec{\mathtt{x}}\mathtt{))\ (}\vec{E}\mathtt{)}\right\}\ Q}{(SPEC, FSPEC) \vdash P \ \left\{\mathtt{y} \leftarrow \mathtt{g(}\vec{E}\mathtt{)}\right\}\ Q} \qquad \begin{array}{r}\vec{E} : \vec{\sigma},\\ \vec{\sigma} \leq \vec{\mathtt{S}}\end{array}$$

[fcall-b] $$\frac{(SPEC, FSPEC) \vdash P \ \left\{\mathtt{y} \leftarrow \mathtt{(fun\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{)\ f(}\vec{\mathtt{x}}\mathtt{))\ (}\vec{E}\mathtt{)}\right\}\ Q}{(SPEC, FSPEC) \vdash P \ \left\{\mathtt{y} \leftarrow \mathtt{f(}\vec{E}\mathtt{)}\right\}\ Q} \qquad \begin{array}{r}\vec{E} : \vec{\sigma},\\ \vec{\sigma} \leq \vec{\mathtt{S}}\end{array}$$

[comb] $$\frac{\begin{array}{c}(SPEC, FSPEC) \vdash R_1 \,\&\, \cdots \,\&\, R_n \ \{\mathtt{y} \leftarrow E_0\}\ Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}]\\ (SPEC, FSPEC) \vdash P \ \{\mathtt{v}_1 \leftarrow E_1\}\ (R_1[\mathtt{v}_1/\mathtt{x}_1])[\vec{\mathtt{x}}/\vec{\mathtt{z}}],\\ \vdots\\ (SPEC, FSPEC) \vdash P \ \{\mathtt{v}_n \leftarrow E_n\}\ (R_n[\mathtt{v}_n/\mathtt{x}_n])[\vec{\mathtt{x}}/\vec{\mathtt{z}}]\end{array}}{(SPEC, FSPEC) \vdash P \ \left\{\mathtt{y} \leftarrow \mathtt{(fun\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{)\ } E_0\mathtt{)\ (}E_1, \ldots, E_n\mathtt{)}\right\}\ Q} \qquad \begin{array}{r}\vec{\mathtt{z}} \text{ fresh}\\ \mathtt{y} \notin \vec{\mathtt{x}}\end{array}$$

[if] $$\frac{\begin{array}{c}(SPEC, FSPEC) \vdash P \ \{\mathtt{v} : \mathtt{Bool} \leftarrow E_1\}\ \text{true},\\ (SPEC, FSPEC) \vdash P\&R_1 \ \{\mathtt{v} : \mathtt{Bool} \leftarrow E_1\}\ \mathtt{v} = \text{true},\\ (SPEC, FSPEC) \vdash P\&R_1 \ \{\mathtt{y} \leftarrow E_2\}\ Q,\\ (SPEC, FSPEC) \vdash P\&R_2 \ \{\mathtt{v} : \mathtt{Bool} \leftarrow E_1\}\ \mathtt{v} = \text{false},\\ (SPEC, FSPEC) \vdash P\&R_2 \ \{\mathtt{y} \leftarrow E_3\}\ Q,\\ (SPEC, FSPEC) \vdash P\&R_3 \ \{\mathtt{y} \leftarrow E_2\}\ Q,\\ (SPEC, FSPEC) \vdash P\&R_3 \ \{\mathtt{y} \leftarrow E_3\}\ Q,\\ SPEC \vdash (R_1|R_2|R_3) = \text{true}\end{array}}{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{fi}\}\ Q}$$

[erratic] $$\frac{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E_1\}\ Q,\ (SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E_2\}\ Q}{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E_1 \sqcup E_2\}\ Q}$$

[angelic] $$\frac{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E_1\}\ Q,\ (SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E_2\}\ Q}{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E_1 \bigtriangledown E_2\}\ Q}$$

[isDef] $$\frac{(SPEC, FSPEC) \vdash P \ \{\mathtt{v} \leftarrow E\}\ \text{true}}{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} : \mathtt{Bool} \leftarrow \mathtt{isDef?(}E\mathtt{)}\}\ \mathtt{y} = \text{true}}$$

[conseq] $$\frac{\begin{array}{c}SPEC \vdash P \Rightarrow P_1,\\ (SPEC, FSPEC) \vdash P_1 \ \{\mathtt{y} \leftarrow E\}\ Q_1,\\ SPEC \vdash Q_1 \Rightarrow Q\end{array}}{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E\}\ Q} \qquad \begin{array}{r}P, P_1, Q_1, Q\\ \text{sub.-con.}\end{array}$$

[equal] $$\frac{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ \mathtt{y} = N}{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ M[\mathtt{y}/\mathtt{z}] = M[N/\mathtt{z}]} \qquad \mathtt{z} : \mathtt{T}$$

[carry] $$\frac{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E\}\ Q}{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E\}\ P \,\&\, Q}$$

[rename] $$\frac{(SPEC, FSPEC) \vdash P[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \ \{\mathtt{y}[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \leftarrow E[\vec{\mathtt{z}}/\vec{\mathtt{x}}]\}\ Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}]}{(SPEC, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E\}\ Q} \qquad \begin{array}{r}\vec{\mathtt{x}} : \vec{\mathtt{T}}\\ \vec{\mathtt{z}} : \vec{\mathtt{T}} \text{ fresh}\end{array}$$

Figure 6.2: Inference rules for verification of NOAL expressions.

If the boolean expression E_1 is deterministic, let the assertion R_3 be "false" and let R_2 be $\neg R_1$. The assertion R_1 must then characterize the value of E_1 in the following sense. If the desired pre-condition and R_1 both hold, then the only possible result of E_1 is *true*, otherwise if the desired pre-condition and $\neg R_1$ both hold, then the only possible result of E_1 is *false*. Then one has to prove that the desired pre-condition and R_1 together are strong enough to show that the desired post-condition holds on the results of E_2 (the "true" arm) and that the desired pre-condition and $\neg R_1$ are strong enough to make the post-condition hold on the result of E_3 (the "false" arm). Since R_3 is "false" and $R_2 = \neg R_1$, the other hypotheses follow trivially.

If the boolean expression E_1 has as possible results both *true* and *false* (e.g., if E_1 is `true [] false`), then let R_1 and R_2 be "false" and let R_3 be "true." One must then show that E_1 terminates and that the possible results of both E_2 and E_3 satisfy the desired post-condition when the desired pre-condition holds. The other hypotheses follow trivially.

In general, the assertion R_1 should characterize when E_1 has *true* as its only possible result, R_2 should characterize when E_1 has *false* as its only possible result, and R_3 should characterize when E_1 is nondeterministic. Then one has to show that E_1 terminates, that in each case the post-condition follows, and that all cases are covered. One shows that all cases are covered by showing that $(R_1|R_2|R_3) = \text{true}$.

- The inference rule [erratic] says that the desired post-condition must follow from the desired pre-condition for each expression.
- The inference rule [angelic] is analogous to the rule [erratic]. Although this rule is sound, it fails to capture all the semantics of angelic choice in NOAL. That is, an angelic choice expression where only one subexpression might fail to terminate would still terminate, but our rule requires that both subexpressions terminate. This incompleteness is caused by the inability of the logic to describe the possible results of an expression separately from its termination.
- The inference rule [isDef] says that to prove that an `isDef?` expression halts (with value *true*), one must prove that all the possible results of the argument expression are proper.
- The general inference rule [conseq] is standard for Hoare logics, where it is often called the "rule of consequence" [Hoa69]. It allows one to use a stronger pre-condition and a weaker post-condition. The implications that appear in the hypothesis must be provable from the traits of the referenced specification, using the proof rules and axioms of those traits. Furthermore they must be subtype-constraining so that the implications also are valid in environments that use subtyping.

  Note that the result identifier `y` must not appear free in P or P_1, since otherwise the triples would not be well-formed.
- The inference rule [equal] allows one to draw subtype-constraining conclusions from equations in post-conditions. The rule would also be valid if the post-condition appearing in the hypothesis were reversed to read "$N =$ `y`." The notation $M[\mathtt{y}/\mathtt{z}]$ means M with all free occurrences of `z` replaced by `y`. Non subtype-constraining assertions can only be introduced into a proof by the [ident] rule.

  The ability to draw subtype-constraining conclusions from an equation in a post-condition is sometimes necessary when one wishes to weaken a post-condition for further use in a proof, because most other rules will require subtype-constraining assertions.
- The inference rule [carry] allows one to carry assertions from the pre-condition into the post-condition. Pre-Conditions are preserved by expressions, because NOAL is applicative.
- The inference rule [rename] allows one to consistently rename identifiers to identifiers of the same nominal type.

A proof in this Hoare logic may also use formulas that are provable from the traits of the referenced specification. (Such formulas are used as hypotheses in the rules [conseq] and [if].) These traits always include the trait Bool and the equality trait. The equality trait allows one to interpret the relation "=" that appears in terms as a congruence relation.

A *proof* of a Hoare-triple H for $(SPEC, FSPEC)$ is a list, where the last line in the list is the formula $(SPEC, FSPEC) \vdash H$ and each line in the list is either:

- a formula of the form $SPEC \vdash Q$, where Q is a $SIG(SPEC)$-assertion that is provable from the traits of $SPEC$, or
- a formula of the form

$$(SPEC, FSPEC) \vdash P' \ \{\mathtt{y} \leftarrow E'\} \ Q',$$

  where $P' \ \{\mathtt{y} \leftarrow E'\} \ Q'$ is a Hoare-triple for $(SPEC, FSPEC)$, and the formula either is an axiom or follows from some previous lines by the inference rules.

6.2 NOAL Program Verification

A method for the verification of NOAL programs is given in this section, along with several examples. This method does not address modularity issues, such as adding a new type to a program; such issues are discussed in Section 6.4.

Program verification compares the meaning of a program against the program's specification. A NOAL program specification is simply a function specification (see Chapter 3). This is shown schematically in Figure 6.3. The verification of the program

$$\vec{F}\ \mathtt{;\ program\ (}\vec{\mathtt{x}}\mathtt{:}\vec{\mathtt{S}}\mathtt{):T\ =}\ E$$

against the specification in Figure 6.3 consists of proving the Hoare-triple $R \ \{\mathtt{v} : \mathtt{T} \leftarrow E\} \ Q$, where `v` is

fun generalProg($\vec{\mathtt{x}} : \vec{\mathtt{S}}$) **returns**(v:T)
requires R
ensures Q

Figure 6.3: General form of a program specification

the formal result identifier from the program specification, R is the program specification's pre-condition, and Q is its post-condition. That is one must show $(SPEC, FSPEC) \vdash R\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$, where $SPEC$ is a set of type specifications that includes at least all the types in $\vec{\mathtt{S}}$, T, the types explicitly mentioned in $\vec{F}$ and E, and the types used indirectly by the above, and $FSPEC$ contains specifications for the functions in $\vec{F}$. For simplicity, the formals of the ⟨heading⟩ of the program's ⟨program expr⟩ must match the formals of the specification exactly.

The method for verification of a NOAL program is to divide and conquer by first specifying and verifying the recursive function definitions that appear in the program. Then one uses the Hoare logic presented above to prove the desired Hoare-triple, using the specifications of the recursively defined functions as axioms.

Two steps are required to verify a system of NOAL function definitions. First, one shows that for each function f,

$$(SPEC, FSPEC) \vdash \mathrm{Pre}(f, \vec{\mathtt{S}'})\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ \mathrm{Post}(f, \vec{\mathtt{S}'})$$

follows from the proof rules, where E is the body of f, $\mathtt{y} : \mathtt{T}$ is the formal result identifier from the specification of f, $\mathrm{Pre}(f, \vec{\mathtt{S}'})$ is the pre-condition from the specification of f in $FSPEC$, and $\mathrm{Post}(f, \vec{\mathtt{S}'})$ is its post-condition. During this proof one can use the axiom scheme [fcall-a], which assumes that each recursively defined function meets its specification. This allows one to prove the partial correctness of function bodies containing recursive calls. The second step is to prove that each function terminates whenever it is called with arguments that model its pre-condition. This step is necessary, since otherwise one could implement a recursive function specification with a body that simply called itself recursively. It is beyond the scope of this report to provide a method for reasoning about termination. Unfortunately, reasoning about recursion in the presence of NOAL's angelic choice operator is non-trivial.

Example 6.2.1. As an example of program verification, consider the specification of *is2in* given in Figure 3.8 and the program

```
program (s:IntSet):Bool = elem(s,2).
```

The verification of this program consists in showing the following formula:

$$\mathtt{IntSet} \vdash \text{true}\ \{\mathtt{b} : \mathtt{Bool} \leftarrow \mathtt{elem(s,2)}\}\ \mathtt{b} = (2 \in \mathtt{s}). \tag{6.3}$$

The specification of `IntSet` is all that is needed for this example, since no other types are mentioned. However, the idea is that this program will satisfy the specification of *is2in* with respect to a set of type specifications that includes other subtypes of `IntSet`, such as `Interval`.

The proof of the above formula will proceed with the assumption that

$$\mathtt{IntSet} \vdash \text{true}\ \{\mathtt{v2} : \mathtt{Int} \leftarrow \mathtt{2}\}\ \mathtt{v2} = 2. \tag{6.4}$$

so that the desugaring for integer literals is not needed.

Proof: By the inference rule [mp-b], it suffices to show that

$$\vdash \text{true}\ \{\mathtt{b} \leftarrow \mathit{combo}\}\ \mathtt{b} = (2 \in \mathtt{s}), \tag{6.5}$$

where *combo* is the expression

```
(fun (s:IntSet, i:Int) elem(s,i)) (s,2).
```

To show the above formula, one must use the rule [comb]. This generates the following subgoals.

$$\begin{array}{ll} & ((2 \in \mathtt{s}) = (2 \in \mathtt{t})) \,\&\, (\mathtt{i} = 2) \\ \vdash & \{\mathtt{b} \leftarrow \mathtt{elem(s,i)}\} \\ & \mathtt{b} = (2 \in \mathtt{t}) \end{array} \tag{6.6}$$

$$\vdash \text{true}\ \{\mathtt{v1} \leftarrow \mathtt{s}\}\ (2 \in \mathtt{v1}) = (2 \in \mathtt{s}) \tag{6.7}$$

$$\vdash \text{true}\ \{\mathtt{v2} \leftarrow \mathtt{2}\}\ \mathtt{v2} = 2. \tag{6.8}$$

The fresh identifier $\mathtt{t} : \mathtt{IntSet}$ is used to refer to the `s` of the outer scope from within the function abstract. As demanded by the [comb] rule, the post-condition of the second goal is derived from "$((2 \in \mathtt{s}) = (2 \in \mathtt{t}))$" by first substituting `v1` for `s` and then `s` for `t`. The assertion "$(2 \in \mathtt{s}) = (2 \in \mathtt{t})$" is subtype-constraining, because "=" is only used between boolean terms.

The first subgoal above follows from the axiom for the `elem` operation in the specification of `IntSet`, and some applications of the rules [conseq] and [carry]. (In what follows the axioms are used to derive the first subgoal, instead of generating more subgoals and working back to the axioms, as above.) In detail, the axiom [mp-a] for the `elem` operation of `IntSet` is:

$$\vdash \text{true}\ \{\mathtt{b} \leftarrow \mathtt{elem(s,i)}\}\ \mathtt{b} = (\mathtt{i} \in \mathtt{s}). \tag{6.9}$$

By the traits of `IntSet` (i.e., by the axioms for the Booleans) one has:

$$\vdash (((2 \in \mathtt{s}) = (2 \in \mathtt{t})) \,\&\, (\mathtt{i} = 2)) \Rightarrow \text{true}. \tag{6.10}$$

So by the inference rule [conseq], the following holds.

$$\begin{array}{ll} & ((2 \in \mathtt{s}) = (2 \in \mathtt{t})) \,\&\, (\mathtt{i} = 2) \\ \vdash & \{\mathtt{b} \leftarrow \mathtt{elem(s,i)}\} \\ & \mathtt{b} = (\mathtt{i} \in \mathtt{s}) \end{array} \tag{6.11}$$

By the inference rule [carry], the pre-condition of the above can be carried into the post-condition.

$$\begin{array}{ll} & (2 \in \mathtt{s}) = (2 \in \mathtt{t}) \,\&\, \mathtt{i} = 2 \\ & \{\mathtt{b} \leftarrow \mathtt{elem(s,i)}\} \\ \vdash & ((2 \in \mathtt{s}) = (2 \in \mathtt{t})) \,\&\, (\mathtt{i} = 2) \\ & \,\&\, (\mathtt{b} = \mathtt{i} \in \mathtt{s}) \end{array} \tag{6.12}$$

By the traits of `IntSet` (i.e., the axioms for equality) one has

$$\vdash \quad (((2 \in \mathtt{s}) = (2 \in \mathtt{t})) \,\&\, (\mathtt{i} = 2) \,\&\, (\mathtt{b} = (\mathtt{i} \in \mathtt{s}))) \\ \Rightarrow (\mathtt{b} = (2 \in \mathtt{t})). \tag{6.13}$$

So by a final application of [conseq] gives the first subgoal of the [comb] rule.

To show the second subgoal of the [comb] rule, one uses the axiom scheme [ident], which gives:

$$\vdash \text{true } \{\mathtt{v1} \leftarrow \mathtt{s}\} \ \mathtt{v1} = \mathtt{s}. \tag{6.14}$$

Then one uses the rule [equal] to derive the second goal.

The third subgoal of the [comb] rule holds by assumption. Thus the conclusion holds. ∎

Example 6.2.2. The way that the logic handles explicit use of subtyping is shown in the proof of the following formula.

$$(\mathrm{II}, \mathit{is2in}) \vdash \begin{array}{l} \text{true} \\ \{\mathtt{b} \leftarrow \mathit{is2in}(\mathtt{create(Interval,1,3)})\} \\ \mathtt{b} = \text{true} \end{array} \tag{6.15}$$

Recall that II combines `IntSet` and `Interval`. In this case both types are mentioned (`IntSet` in the signature of *is2in*).

The axiom [funcall-a] derived from the specification of *is2in* in Figure 3.8 is also assumed. Furthermore, to concentrate on the interesting part of the proof, it is assumed that the following holds (see the specification of `Interval`).

$$(\mathrm{II}, \mathit{is2in}) \vdash \begin{array}{l} \text{true} \\ \{\mathtt{v1} \leftarrow \mathtt{create(Interval,1,3)}\} \\ \mathtt{v1} == [1,3] \end{array} \tag{6.16}$$

Proof: Since the expression in question consists of a function call one must use the rule [fcall-b]. This gives the following goal.

$$\vdash \begin{array}{l} \text{true} \\ \left\{\mathtt{b} \leftarrow \begin{array}{l} \mathtt{(fun\ (s:IntSet)}\ \mathit{is2in}\mathtt{(s))} \\ \mathtt{(create(Interval,1,3))} \end{array}\right\} \\ \mathtt{b} = \text{true} \end{array} \tag{6.17}$$

Since the expression in the above goal is a combination, the [comb] rule is used to give the following subgoals:

$$\vdash \begin{array}{l} \mathtt{s} == [1,3] \\ \{\mathtt{b} \leftarrow \mathit{is2in}\mathtt{(s)}\} \\ \mathtt{b} = \text{true} \end{array} \tag{6.18}$$

$$\vdash \begin{array}{l} \text{true} \\ \{\mathtt{v1} \leftarrow \mathtt{create(Interval,1,3)}\} \\ \mathtt{v1} == [1,3] \end{array} \tag{6.19}$$

The above goals are similar to the operational idea of substituting the abstract value of the argument ([1,3])

fun testFor(i:Int, s1,s2: IntSet) **returns**(j:Int)
requires (i ∈ s1) & (¬(isEmpty(s1 ∩ s2)))
ensures (j ∈ s1) & (j ∈ s2)

Figure 6.4: Specification of the function *testFor*.

in the specification of *is2in*. The identifier `s` has nominal type `IntSet`, as that is the nominal argument type of *is2in*, while `v1` has nominal type `Interval`. Furthermore, the assertion "`s` == [1,3]" is interpreted by the trait-function "#==#" with signature

$$\mathtt{IntSet}, \mathtt{Interval} \rightarrow \mathtt{Bool}.$$

The first hypothesis of the [comb] rule follows from the axiom scheme [fcall-a] for *is2in*, and the inference rules [conseq] and [carry]. The axiom for *is2in* is:

$$\vdash \text{true } \{\mathtt{b} \leftarrow \mathit{is2in}\mathtt{(s)}\} \ \mathtt{b} = 2 \in \mathtt{s} \tag{6.20}$$

From the traits of II one can show that

$$\vdash (\mathtt{s} == [1,3]) \Rightarrow ((2 \in \mathtt{s}) = (2 \in [1,3])). \tag{6.21}$$

Furthermore, it follows from the traits of II that

$$\vdash (2 \in [1,3]) = \text{true}, \tag{6.22}$$

hence

$$\vdash (\mathtt{s} == [1,3]) \Rightarrow ((2 \in \mathtt{s}) = \text{true}). \tag{6.23}$$

Thus the first hypothesis follows from the rules [conseq] (used twice) and [carry].

The second hypothesis was assumed to hold, so the desired result follows. ∎

Example 6.2.3. An example of recursive function verification is provided by verifying the implementation of *inBoth* given in Figure 1.4 against the specification given in Figure 1.6.

Since *inBoth* calls the function *testFor* (see Figure 1.4), it is necessary to specify and verify the implementation of *testFor* as well. The specification of *testFor* is given in Figure 6.4.

During the verification of *inBoth* and *testFor*, the specification of *testFor* is used as an axiom, to establish their partial correctness. The termination of *testFor* follows because at each recursive call the size of the argument `s1` shrinks. The verifications use the specification `IntSet`, since that is the only non-visible type mentioned.

To formally verify the partial correctness of the implementation of *inBoth* one must show the following.

$$(\mathtt{IntSet}, \mathit{testFor}) \vdash \\ \neg(\mathrm{isEmpty}(\mathtt{s1} \cap \mathtt{s2})) \\ \{\mathtt{i} \leftarrow \mathit{testFor}\mathtt{(choose(s1),\ s1,\ s2)}\} \\ (\mathtt{i} \in \mathtt{s1}) \,\&\, (\mathtt{i} \in \mathtt{s2}) \tag{6.24}$$

Proof: To avoid name clashes with the result identifier, the rule [rename] is used to generate the following

goal.

$$\vdash \begin{array}{l} \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})) \\ \{\texttt{j} \leftarrow \textit{testFor}(\texttt{choose(s1), s1, s2})\} \\ (\texttt{j} \in \texttt{s1}) \,\&\, (\texttt{j} \in \texttt{s2}) \end{array} \quad (6.25)$$

Since the expression in question is a function call, one must use the rule [fcall-b]. This gives a goal with the same pre- and post-conditions as above, but whose body is

```
(fun (i:Int,s1,s2:IntSet)
     testFor(i,s1,s2))
(choose(s1), s1, s2)
```

Since the expression above is a combination, the [comb] rule is used to give the following subgoals,

$$\vdash \begin{array}{l} R_1 \& R_2 \& R_3 \\ \{\texttt{j} \leftarrow \textit{testFor}(\texttt{i,s1,s2})\} \\ (\texttt{j} \in \texttt{t1}) \& (\texttt{j} \in \texttt{t2}) \end{array} \quad (6.26)$$

$$\vdash \begin{array}{l} \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})) \\ \{\texttt{v1} \leftarrow \texttt{choose(s1)}\} \\ \texttt{v1} \in \texttt{s1} \end{array} \quad (6.27)$$

$$\vdash \begin{array}{l} \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})) \\ \{\texttt{v2} \leftarrow \texttt{s1}\} \\ (\texttt{v2} == \texttt{s1}) \& (\neg(\text{isEmpty}(\texttt{v2} \cap \texttt{s2}))) \end{array} \quad (6.28)$$

$$\vdash \begin{array}{l} \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})) \\ \{\texttt{v3} \leftarrow \texttt{s2}\} \\ (\texttt{v3} == \texttt{s2}) \& (\neg(\text{isEmpty}(\texttt{s1} \cap \texttt{v3}))) \end{array} \quad (6.29)$$

where

$$\begin{array}{rcl} R_1 & = & (\texttt{i} \in \texttt{s1}) \\ R_2 & = & ((\texttt{s1} == \texttt{t1}) \& (\neg(\text{isEmpty}(\texttt{s1} \cap \texttt{t2})))) \\ R_3 & = & ((\texttt{s2} == \texttt{t2}) \& (\neg(\text{isEmpty}(\texttt{t1} \cap \texttt{s2})))). \end{array}$$

(For fresh variables, `t1`, and `t2`, are used to replace `s1`, and `s2`, respectively.)

The first subgoal is shown as follows. By the traits of `IntSet`, if follows that

$$\vdash \begin{array}{l} (R_1 \& R_2 \& R_3) \\ \Rightarrow ((\texttt{i} \in \texttt{s1}) \& (\neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})))) \end{array} \quad (6.30)$$

so by [conseq] and the axiom [funcall-a] for *testFor* it suffices to show that:

$$\vdash \begin{array}{l} R_1 \& R_2 \& R_3 \\ \{\texttt{j} \leftarrow \textit{testFor}(\texttt{i,s1,s2})\} \\ (\texttt{j} \in \texttt{s1}) \& (\texttt{j} \in \texttt{s2}) \end{array} \quad (6.31)$$

To show the above, one can use [carry] to bring the R_i into the post-condition.

$$\vdash \begin{array}{l} R_1 \& R_2 \& R_3 \\ \{\texttt{j} \leftarrow \textit{testFor}(\texttt{i,s1,s2})\} \\ (\texttt{j} \in \texttt{s1}) \& (\texttt{j} \in \texttt{s2}) \& R_1 \& R_2 \& R_3 \end{array} \quad (6.32)$$

The first subgoal follows by the traits of `IntSet` and the rule [conseq]. Note that the R_i are subtype-constraining.

The second subgoal is shown as follows. By the traits of `IntSet` it follows that

$$\vdash \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})) \Rightarrow \neg(\text{isEmpty}(\texttt{s1})) \quad (6.33)$$

and since the assertions in the above implication are subtype-constraining, by [conseq] it suffices to show that:

$$\vdash \neg(\text{isEmpty}(\texttt{s1})) \;\; \{\texttt{v1} \leftarrow \texttt{choose(s1)}\} \;\; \texttt{v1} \in \texttt{s1} \quad (6.34)$$

By the rule [rename] it suffices to show the following

$$\vdash \neg(\text{isEmpty}(\texttt{s})) \;\; \{\texttt{i} \leftarrow \texttt{choose(s)}\} \;\; \texttt{i} \in \texttt{s} \quad (6.35)$$

But this last formula is the axiom [mp-a] for `choose`.

The third and fourth subgoals follow from the rules [ident], [equal], and [conseq]. ∎

To formally verify the partial correctness of the implementation of *testFor*, one must show that:

$$(\texttt{IntSet}, \textit{testFor}) \vdash \begin{array}{l} (\texttt{i} \in \texttt{s1}) \& \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})) \\ \{\texttt{j} \leftarrow \textit{Body}\} \\ (\texttt{j} \in \texttt{s1}) \& (\texttt{j} \in \texttt{s2}) \end{array} \quad (6.36)$$

where *Body* is the body of *testFor*.

Proof: Since the body of *testFor* is an `if` expression, one must use the rule [if]. This gives the goals listed as hypotheses for the rule [if], where P is the pre-condition above, Q is the post-condition above, `y` is the identifier `j`, E_1 is the expression `elem(s2,i)`, E_2 is the expression `i`, E_3 is the expression

```
testFor(choose(remove(s1,i)),
        remove(s1,i), s2)
```

R_1 is the assertion "$(\texttt{i} \in \texttt{s2})$", R_2 is $\neg R_1$, and R_3 is "false." The R_i characterize the various cases exhaustively by definition. Hence the last subgoal of the [if] rule holds:

$$\vdash (R_1 | R_2 | R_3) = \text{true}. \quad (6.37)$$

Furthermore, since R_3 is false, the other subgoals involving R_3 hold trivially, using [carry] and [conseq]. The subgoal

$$\vdash \begin{array}{l} (\texttt{i} \in \texttt{s1}) \& \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})) \\ \{\texttt{v} \leftarrow \texttt{elem(s2,i)}\} \\ \text{true} \end{array} \quad (6.38)$$

in which one shows termination of the test is proved by using [rename] and the axiom [mp-a] for `elem` and [conseq]. Thus the remaining subgoals center around the true and false cases. For the "true" case, one must show the following.

$$\vdash \begin{array}{l} (\texttt{i} \in \texttt{s1}) \,\&\, \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})) \,\&\, (\texttt{i} \in \texttt{s2}) \\ \{\texttt{v} \leftarrow \texttt{elem(s2,i)}\} \\ \texttt{v} = \text{true} \end{array} \quad (6.39)$$

$$\vdash \begin{array}{l} (\texttt{i} \in \texttt{s1}) \,\&\, \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})) \,\&\, (\texttt{i} \in \texttt{s2}) \\ \{\texttt{j} \leftarrow \texttt{i}\} \\ (\texttt{j} \in \texttt{s1}) \,\&\, (\texttt{j} \in \texttt{s2}) \end{array} \quad (6.40)$$

The first of the above goals follows by using [rename] and the axiom [mp-a] for `elem` and [conseq]. The second goal for the "true" case follows from the rules [ident] and [conseq]. For the "false" case one must show that

$$\begin{array}{ll} & (\mathtt{i} \in \mathtt{s1}) \& \neg(\mathrm{isEmpty}(\mathtt{s1} \cap \mathtt{s2})) \& \neg(\mathtt{i} \in \mathtt{s2}) \\ \vdash & \{\mathtt{v} \leftarrow \mathtt{elem(s2,i)}\} \\ & \mathtt{v} = \mathrm{false} \end{array} \tag{6.41}$$

$$\begin{array}{ll} & (\mathtt{i} \in \mathtt{s1}) \& \neg(\mathrm{isEmpty}(\mathtt{s1} \cap \mathtt{s2})) \& \neg(\mathtt{i} \in \mathtt{s2}) \\ \vdash & \{\mathtt{j} \leftarrow E_3\} \\ & (\mathtt{j} \in \mathtt{s1}) \& (\mathtt{j} \in \mathtt{s2}) \end{array} \tag{6.42}$$

The first of these goals follows as in the "true" case. Thus it suffices to show the second goal of the "false" case.

The second goal of the "false" case follows from the [fcall-b], [comb], the axiom [funcall-a] for *testFor*, the axioms [mp-a] for `choose` and `remove`, [rename], and [conseq]. Using the rule [fcall-b] one rewrites the expression E_3 as the following combination.

```
(fun (i:Int, s1,s2:IntSet)
     testFor(i,s1,s2))
 (choose(remove(s1,i)), remove(s1,i), s2)
```

The rule [comb] generates the following subgoals.

$$\begin{array}{ll} & (\mathtt{i} \in \mathtt{t1}) \\ & \quad \& (\neg(\mathrm{isEmpty}(\mathtt{s1} \cap \mathtt{t2})) \\ \vdash & \quad \&(\mathtt{s1} == \mathrm{delete}(\mathtt{t1}, \mathtt{t0}))) \\ & \quad \& (\mathtt{s2} == \mathtt{t2}) \\ & \{\mathtt{j} \leftarrow \mathit{testFor}\mathtt{(i,s1,s2)}\} \\ & (\mathtt{j} \in \mathtt{t1}) \,\&\, (\mathtt{j} \in \mathtt{t2}) \end{array} \tag{6.43}$$

$$\begin{array}{ll} & (\mathtt{i} \in \mathtt{s1}) \\ & \quad \& \neg(\mathrm{isEmpty}(\mathtt{s1} \cap \mathtt{s2})) \\ \vdash & \quad \& \neg(\mathtt{i} \in \mathtt{s2}) \\ & \{\mathtt{v1} \leftarrow \mathtt{choose(remove(s1,i))}\} \\ & \mathtt{v1} \in \mathtt{s1} \end{array} \tag{6.44}$$

$$\begin{array}{ll} & (\mathtt{i} \in \mathtt{s1}) \\ & \quad \& \neg(\mathrm{isEmpty}(\mathtt{s1} \cap \mathtt{s2})) \\ \vdash & \quad \& \neg(\mathtt{i} \in \mathtt{s2}) \\ & \{\mathtt{v2} \leftarrow \mathtt{remove(s1,i)}\} \\ & \quad \neg(\mathrm{isEmpty}(\mathtt{v2} \cap \mathtt{s2})) \\ & \quad \&(\mathtt{v2} == \mathrm{delete}(\mathtt{s1}, \mathtt{i})) \end{array} \tag{6.45}$$

$$\begin{array}{ll} & (\mathtt{i} \in \mathtt{s1}) \\ & \quad \& \neg(\mathrm{isEmpty}(\mathtt{s1} \cap \mathtt{s2})) \\ \vdash & \quad \& \neg(\mathtt{i} \in \mathtt{s2}) \\ & \{\mathtt{v3} \leftarrow \mathtt{s2}\} \\ & \mathtt{v3} == \mathtt{s2} \end{array} \tag{6.46}$$

The fresh identifiers $\vec{\mathtt{z}}$ demanded by [comb] are `t0`, `t1`, and `t2`, which refer to the nonlocal bindings of `i`, `s1`, and `s2` from within the function abstract. Thus the hypothesis of the first subgoal says that `i` is in the original set `s1`, and the new `s1` argument intersects the original set `s2` in addition to being formed from the original `s1` by deleting the original `i`, and that the new `s2` has the same elements as the original `s2`. To prove the second subgoal, one must show that `remove(s1,i)` is not empty. This follows from the pre-condition, since there `s1` intersects `s2` and `i` is not in `s2`. To prove the third subgoal one must show that removing `i` from `s1` does not leave the intersection with `s2` empty, but this will follow from the pre-condition of that goal, since `i` is not in `s2` and thus not in the intersection. ∎

6.3 Soundness of Hoare-style Verification for NOAL

The soundness of the Hoare logic for NOAL and the method for verifying NOAL programs given above are proved in this section. The logic and verification method reach valid conclusions when the specification's $\leq$ relation is a subtype relation. The key to the soundness proof is the existence of an algebra and a simulation that are guaranteed by the definition of subtype relations.

There are several lemmas used in the soundness proof. The first set of lemmas shows that assertions can be lifted to supertypes and remain valid. The second set of lemmas shows that truth is preserved by simulation for subtype-constraining assertions. The third set of lemmas shows that subtype-constraining assertions provable from the traits of a specification are valid even in environments that admit subtyping. Finally the fourth set of lemmas establishes soundness by induction on the length of proof in the Hoare logic.

6.3.1 Assertions can be Lifted

The main lemma of this section states that if a formula is valid, then it remains valid when one changes the types of some of the identifiers to supertypes of their initial types. The proviso is that the formula must still sort-check when the types of the identifiers are changed. It is technically convenient to regard the process as moving renamings from the formula into the environment; that is, if the formula (renamed with subtypes for the identifiers), is modeled by the environment, then the environment (extended by binding the previous values to identifiers at the supertype) models the unrenamed formula.

In essence, the main lemma holds because the result of a specification function does not depend on the nominal type of an argument. This is shown formally in the following lemma.

The notation $\eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]$ means the environment η extended by simultaneously binding the identifiers $\mathtt{x}_i$ to $\eta(\mathtt{v}_i)$, for all i.

Lemma 6.3.1. Let *SPEC* be a set of type specifications. Let $\leq$ be the presumed subtype relation of $SIG(SPEC)$. Let C be a *SPEC*-algebra. Let X be a set of identifiers containing $\vec{\mathtt{x}} : \vec{\mathtt{T}}$. Let Y be a set of identifiers containing $\vec{\mathtt{v}} : \vec{\mathtt{S}}$ such that $\vec{\mathtt{S}} \leq \vec{\mathtt{T}}$ and $Y \cup \{\mathtt{x}_i : \mathtt{T}_i\}_i \supseteq X$. Let Q be a $SIG(SPEC)$-term with free identifiers from X. Let $\eta : Y \to |C|$ be a proper $SIG(SPEC)$-environment.

Then

$$\overline{\eta}[\![Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!] = \overline{\eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]}[\![Q]\!].$$

Proof: (by induction on the structure of terms.)
For the basis, suppose that Q is a nullary specification function or an identifier. In the former case, the value of Q does not depend on the environment. In the latter case, the result follows because of the way the environments are set up. That is, if Q is $\mathtt{x}$, then

$$Q[\mathtt{v}/\mathtt{x}] = \mathtt{v} \quad (6.47)$$
$$\overline{\eta}[\![\mathtt{v}]\!] = \eta(\mathtt{v}) \quad (6.48)$$
$$\overline{\eta[\eta(\mathtt{v})/\mathtt{x}]}[\![Q]\!] = \eta(\mathtt{v}). \quad (6.49)$$

For the inductive step, assume that the lemma holds for all subterms of Q. There are two cases.

- Suppose Q has the form $\mathrm{f}(\vec{E})$, where "f" is a specification function. Then by the definition of the extended environment and the inductive hypothesis:

$$\overline{\eta}[\![\mathrm{f}(\vec{E})[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!] = \mathrm{f}^C(\overline{\eta}[\![\vec{E}[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!]) \quad (6.50)$$
$$= \mathrm{f}^C(\overline{\eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]}[\![\vec{E}]\!]) \quad (6.51)$$
$$= \overline{\eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]}[\![\mathrm{f}(\vec{E})]\!]. \quad (6.52)$$

- Suppose Q has the form $E_1 = E_2$. By definition of substitution, the extended environment and the inductive hypothesis:

$$\overline{\eta}[\![(E_1 = E_2)[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!]$$
$$= \overline{\eta}[\![E_1[\vec{\mathtt{v}}/\vec{\mathtt{x}}] = E_2[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!] \quad (6.53)$$
$$= \begin{cases} \mathit{true} & \text{if } \overline{\eta}[\![E_1[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!] \\ & \quad = \overline{\eta}[\![E_2[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!] \\ \mathit{false} & \text{otherwise} \end{cases} \quad (6.54)$$
$$= \begin{cases} \mathit{true} & \text{if } \overline{\eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]}[\![E_1]\!] \\ & \quad = \overline{\eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]}[\![E_2]\!] \\ \mathit{false} & \text{otherwise} \end{cases} \quad (6.55)$$
$$= \overline{\eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]}[\![E_1 = E_2]\!]. \quad (6.56)$$

∎

The following lemma is the "main lemma" of this section.

Lemma 6.3.2. Let $SPEC$ be a set of type specifications. Let $\leq$ be the presumed subtype relation of $SIG(SPEC)$. Let C be a $SPEC$-algebra. Let X be a set of identifiers containing $\vec{\mathtt{x}} : \vec{\mathtt{T}}$. Let Y be a set of identifiers containing $\vec{\mathtt{v}} : \vec{\mathtt{S}}$ such that $\vec{\mathtt{S}} \leq \vec{\mathtt{T}}$ and $Y \cup \{\mathtt{x}_i : \mathtt{T}_i\}_i \supseteq X$. Let Q be a $SIG(SPEC)$-assertion with free identifiers from X. Let $\eta : Y \to |C|$ be a proper $SIG(SPEC)$-environment.
If $(C, \eta) \models Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$, then $(C, \eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]) \models Q$.

Proof: Suppose $(C, \eta) \models Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$. By definition, $\overline{\eta}[\![Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!] = \mathit{true}$. By the above lemma,

$$\overline{\eta}[\![Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!] = \overline{\eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]}[\![Q]\!]. \quad (6.57)$$

So by definition, $(C, \eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]) \models Q$. ∎

6.3.2 Simulation is Preserved by Subtype-Constraining Assertions

The following lemmas describe the relationship between simulation and validity for subtype-constraining assertions. The first lemma says that simulation is preserved by subtype-constraining terms. The second specializes the first lemma to assertions.
The first lemma is analogous to the fundamental theorem of logical relations for specification functions.
Environments are related pointwise. Given Σ-environments $\eta_B : X \to |B|$ and $\eta_A : X \to |A|$, the notation $\eta_B \mathcal{R} \eta_A$ means that for all sorts $\mathtt{T}$, for all $\mathtt{x} : \mathtt{T} \in X$, $\eta_B(\mathtt{x}) \mathcal{R}_{\mathtt{T}} \eta_A(\mathtt{x})$.

Lemma 6.3.3. Let Σ be a signature. Let C and A be a Σ-algebras. Let X be a set of identifiers. Let $\mathtt{S}$ be a sort of Σ. Let Q be an term with free identifiers from X and nominal sort $\mathtt{S}$.
If Q is subtype-constraining, $\mathcal{R}$ is a Σ-simulation relation between C and A, and $\eta_C : X \to |C|$ and $\eta_A : X \to |A|$ are environments such that $\eta_C \mathcal{R} \eta_A$, then

$$\overline{\eta_C}[\![Q]\!] \mathcal{R}_{\mathtt{S}} \overline{\eta_A}[\![Q]\!].$$

Proof: (by induction on the structure of terms).
For the basis there are two cases. If Q is an identifier $\mathtt{x} : \mathtt{S}$, then by hypothesis $\eta_C(\mathtt{x}) \mathcal{R}_{\mathtt{S}} \eta_A(\mathtt{x})$. If Q is a nullary specification function, then by the substitution property for specification functions, $\overline{\eta_C}[\![Q]\!] \mathcal{R}_{\mathtt{S}} \overline{\eta_A}[\![Q]\!]$.
For the inductive step, assume that the result holds for all subterms of Q. There are also two cases.
If Q has the form $\mathrm{f}(\vec{E})$, then by the inductive hypothesis, for each of the E_i, if the nominal sort of E_i is $\mathtt{S}_i$, then $\overline{\eta_C}[\![E_i]\!] \mathcal{R}_{\mathtt{S}_i} \overline{\eta_A}[\![E_i]\!]$. Thus the result follows by the substitution property for specification functions.
If Q has the form $E_1 = E_2$, then since Q is subtype-constraining, E_1 the nominal sort of E_1 is a visible sort, say $\mathtt{T}$. By the inductive hypothesis, for each of the E_i, $\overline{\eta_C}[\![E_i]\!] \mathcal{R}_{\mathtt{T}} \overline{\eta_A}[\![E_i]\!]$. Since $\mathtt{T}$ is visible, $\mathcal{R}_{\mathtt{T}}$ is the identity on $\mathtt{T}$. Since there can be no subtypes of a visible type, for each of the E_i, $\overline{\eta_C}[\![E_i]\!] = \overline{\eta_A}[\![E_i]\!]$. So if $\overline{\eta_C}[\![E_1]\!] = \overline{\eta_C}[\![E_2]\!]$, then $\overline{\eta_A}[\![E_1]\!] = \overline{\eta_A}[\![E_2]\!]$ and if $\overline{\eta_C}[\![E_1]\!] \neq \overline{\eta_C}[\![E_2]\!]$, then $\overline{\eta_A}[\![E_1]\!] \neq \overline{\eta_A}[\![E_2]\!]$. ∎
An important consequence of the above lemma is that if one environment simulates another, then the same set of subtype-constraining assertions is valid in each.

Lemma 6.3.4. Let Σ be a signature. Let C and A be a Σ-algebras. Let X be a set of identifiers. Let Q be a Σ-assertion with free identifiers from X.
If Q is subtype-constraining, $\mathcal{R}$ is a Σ-simulation relation between C and A, and $\eta_C : X \to |C|$ and $\eta_A : X \to |A|$ are environments such that $\eta_C \mathcal{R} \eta_A$, then $(C, \eta_C) \models Q$ if and only if $(A, \eta_A) \models Q$.

Proof: Suppose Q is subtype-constraining, $\mathcal{R}$ is a Σ-simulation relation between C and A, and $\eta_C : X \to |C|$ and $\eta_A : X \to |A|$ are environments such that $\eta_C \mathcal{R} \eta_A$. Since Q is an assertion, its nominal sort is $\mathtt{Bool}$. So by lemma 6.3.3,

$$\overline{\eta_C}[\![Q]\!] \mathcal{R}_{\mathtt{Bool}} \overline{\eta_A}[\![Q]\!]. \quad (6.58)$$

But $\mathcal{R}_{\mathtt{Bool}}$ is the identity, so either both $\overline{\eta_C}[\![Q]\!]$ and $\overline{\eta_A}[\![Q]\!]$ are *true* or neither is. ∎

6.3.3 Provable and Subtype-Constraining Assertions are Valid

Assertions provable from the traits of a specification are only required to be valid in nominal environments, since that is the "standard definition of satisfaction" for traits. The following lemma shows that subtype-constraining assertions that are provable from a specification are valid in all environments, even those that admit subtyping, provided the presumed subtype relation is really a subtype relation.

The restriction to subtype-constraining assertions is necessary, since properties of a supertype that are not subtype-constraining may not hold for a subtype. An example is the following assertion, where $\mathtt{s} : \mathtt{IntSet}$:

$$\text{isEmpty}(\mathtt{s}) \Rightarrow \mathtt{s} = \{\}. \tag{6.59}$$

The above assertion is provable from the trait IntSet-Trait, which describes the abstract values of type `IntSet`, but it does not hold for environments where `s` may denote a `PSchd` object (since the abstract values of `PSchd` objects are pairs).

Lemma 6.3.5. Let $SPEC$ be a set of type specifications. Let X be a set of identifiers. Let Q be a $SIG(SPEC)$-assertion with free identifiers from X.

If Q is subtype-constraining, $SPEC \vdash Q$, and $\leq$ is a subtype relation on the types of $SPEC$, then for all $SPEC$-algebras C and for all proper $SIG(SPEC)$-environments $\eta_C : X \rightarrow |C|$, $(C, \eta_C) \models Q$.

Proof: Suppose that $SPEC \vdash Q$ and that $\leq$ is a subtype relation.

Let C be a $SPEC$-algebra. Let $\eta_C : X \rightarrow |C|$, be a proper $SIG(SPEC)$-environment.

By definition of subtype relations, there is some $SPEC$-algebra A such that there is a $SIG(SPEC)$-simulation relation, $\mathcal{R}$, between C and A. Construct a nominal environment $\eta_A : X \rightarrow |A|$ so that for each type $\mathtt{T}$ and for each $\mathtt{x} : \mathtt{T} \in X$,

$$\eta_A(\mathtt{x}) \quad \in \quad \mathtt{T}^A \tag{6.60}$$

$$\eta_C(\mathtt{x}) \quad \mathcal{R}_{\mathtt{T}} \quad \eta_A(\mathtt{x}). \tag{6.61}$$

It is always possible to build such a nominal environment, because of the coercion properties of the simulation $\mathcal{R}$, which guarantee that each object of a subtype of $\mathtt{T}$ simulates some object of type $\mathtt{T}$. (That $\eta_C(\mathtt{x}) \in \mathtt{S}^C$ for some $\mathtt{S} \leq \mathtt{T}$ is guaranteed by the definition of a $SIG(SPEC)$-environment.)

Since $SPEC \vdash Q$, and η_A is nominal, by definition of when an algebra satisfies its traits,

$$\overline{\eta_A}[\![Q]\!] = \mathit{true}. \tag{6.62}$$

Hence $(A, \eta_A) \models Q$.

Since Q is subtype-constraining, by Lemma 6.3.4, $(C, \eta_C) \models Q$. ∎

6.3.4 Soundness Theorems

The following lemma is the essential step in proving soundness for the Hoare logic. It says that if some Hoare-triple is provable, then it is valid. Soundness follows directly.

The lemma's proof is by induction on the length of proof in the Hoare logic. The interesting cases are the axiom schemes [mp-a], and the rules [comb] and [carry] since these are the rules where there is a substantial difference from standard Hoare logics. The soundness of the [mp-a] and [comb] rules relies on the simulation that is implicit in a subtype relation. The proof of the rule [carry] relies on the lack of mutation and assignment in NOAL.

Lemma 6.3.6. Let $(SPEC, FSPEC)$ be a pair of type and function specification sets. Let $\leq$ be the presumed subtype relation of $SIG(SPEC)$.

Suppose $\leq$ is a subtype relation on the types of $SPEC$.

Then for all Hoare-triples for $(SPEC, FSPEC)$, if

$$(SPEC, FSPEC) \vdash P \ \{\mathtt{y} : \mathtt{T} \leftarrow E\} \ Q,$$

then

$$(SPEC, FSPEC) \models P \ \{\mathtt{y} : \mathtt{T} \leftarrow E\} \ Q.$$

Proof: (by induction on the length of proof in the Hoare logic.)

Let $P \ \{\mathtt{y} : \mathtt{T} \leftarrow E\} \ Q$ be a Hoare-triple. Suppose that

$$(SPEC, FSPEC) \vdash P \ \{\mathtt{y} : \mathtt{T} \leftarrow E\} \ Q. \tag{6.63}$$

Let X be a set of identifiers such that X contains all the free identifiers of P and E and Q except $\mathtt{y}$.

For each function identifier f in the domain of $SIG(FSPEC)$, let $\mathcal{F}[\![f]\!]$ be a function denotation with signature $SIG(FSPEC)(f)$ such that $\mathcal{F}[\![f]\!]$ satisfies the specification of f with respect to $SPEC$. To find the set of possible results of a NOAL expression with free function identifiers one needs an environment that is defined on the function identifiers in $FSPEC$. Given an environment η over an algebra C, the desired environment is constructed by extending η so that for each function identifier f, $\eta(f) = \mathcal{F}[\![f]\!](C)$. Since this expansion is unique, to avoid notational complications, the expansion is not mentioned below.

By Lemma 5.3.3, the type of each possible result is a subtype of the result identifier of the Hoare-triple.

For the basis, the result must be shown for each of the axiom schemes. For these cases, let C be a $SPEC$-algebra and η_C an appropriate Σ-environment.

- Suppose the proof consists of an instance of the axiom scheme [ident]

  $$\vdash \text{true} \ \{\mathtt{v} \leftarrow \mathtt{x}\} \ \mathtt{v} = \mathtt{x}.$$

  It is trivial that $(C, \eta_C) \models$ true. By definition, $\mathcal{M}[\![\mathtt{x}]\!](C, \eta_C) = \{\eta_C(\mathtt{x})\}$. Since η_C is proper $\eta_C(\mathtt{x})$ is also proper. Therefore, $(C, \eta_C[\eta_C(\mathtt{x})/\mathtt{v}]) \models \mathtt{v} = \mathtt{x}$.

- Suppose the proof consists of an instance of the axiom scheme [bot]

  $$\vdash \text{false} \ \{\mathtt{v} \leftarrow \mathtt{bottom[T]}\} \ \text{true}.$$

  Then the result follows trivially, since η_C does not model the pre-condition "false."

- Suppose the proof consists of an instance of the axiom scheme [mp-a]:

$$\vdash \mathrm{Pre}(\mathtt{g},\vec{\mathtt{S}})\ \ \{\mathtt{y}:\mathtt{T} \leftarrow \mathtt{g}(\vec{\mathtt{x}})\}\ \ \mathrm{Post}(\mathtt{g},\vec{\mathtt{S}}).$$

where $\mathtt{T} = \mathit{ResSort}(\mathtt{g},\vec{\mathtt{S}})$.

Suppose that $(C,\eta_C) \models \mathrm{Pre}(\mathtt{g},\vec{\mathtt{S}})$.

Since $\leq$ is a subtype relation, there is some $SPEC$-algebra A, such that there is a $SIG(SPEC)$-simulation relation, $\mathcal{R}$, between C and A. Construct a nominal environment $\eta_A : X \rightarrow |A|$ so that for each type $\mathtt{U}$ and for each $\mathtt{x}:\mathtt{U} \in X$,

$$\eta_A(\mathtt{x}) \quad \in \quad \mathtt{U}^A \tag{6.64}$$
$$\eta_C(\mathtt{x}) \quad \mathcal{R}_{\mathtt{U}} \quad \eta_A(\mathtt{x}). \tag{6.65}$$

It is always possible to build such a nominal environment, because of the coercion properties of the simulation $\mathcal{R}$, which guarantee that each object of a subtype of $\mathtt{U}$ simulates some object of type $\mathtt{U}$. (That $\eta_C(\mathtt{x}) \in \mathtt{V}^C$ for some $\mathtt{V} \leq \mathtt{U}$ is guaranteed by the definition of a $SIG(SPEC)$-environment.)

Since the assertion $\mathrm{Pre}(\mathtt{g},\vec{\mathtt{S}})$ must be subtype-constraining, by Lemma 6.3.4

$$(A,\eta_A) \models \mathrm{Pre}(\mathtt{g},\vec{\mathtt{S}}). \tag{6.66}$$

By definition of NOAL,

$$\mathcal{M}[\![\mathtt{g}(\vec{\mathtt{x}})]\!](C,\eta_C) \quad = \quad \mathtt{g}^C(\eta_C(\vec{\mathtt{x}})) \tag{6.67}$$
$$\mathcal{M}[\![\mathtt{g}(\vec{\mathtt{x}})]\!](A,\eta_A) \quad = \quad \mathtt{g}^A(\eta_A(\vec{\mathtt{x}})). \tag{6.68}$$

By construction of η_A, $\eta_C(\vec{\mathtt{x}})\ \mathcal{R}_{\vec{\mathtt{S}}}\ \eta_A(\vec{\mathtt{x}})$, and thus by the substitution property of simulation relations (see Figure 6.5):

$$\forall(q \in \mathtt{g}^C(\eta_C(\vec{\mathtt{x}})))\exists(r \in \mathtt{g}^A(\eta_A(\vec{\mathtt{x}})))q\ \mathcal{R}_{\mathtt{T}}\ r. \tag{6.69}$$

Since η_A is a nominal environment, by definition of when an operation satisfies its specification, for all possible results $r \in \mathtt{g}^A(\eta_A(\vec{\mathtt{x}}))$, $r \neq \perp$ and $(A,\eta_A[r/\mathtt{y}]) \models \mathrm{Post}(\mathtt{g},\vec{\mathtt{S}})$. Since $\mathcal{R}_{\mathtt{T}}$ is bistrict, for all $q \in \mathtt{g}^C(\eta_C(\vec{\mathtt{x}}))$, $q \neq \perp$. Finally, since $\mathrm{Post}(\mathtt{g},\vec{\mathtt{S}})$ is subtype-constraining, for each $q \in \mathtt{g}^C(\eta_C(\vec{\mathtt{x}}))$ there is some $r \in \mathtt{g}^A(\eta_A(\vec{\mathtt{x}}))$ such that $q\ \mathcal{R}_{\mathtt{T}}\ r$, and since all such r satisfy the post-condition, $(C,\eta_C[q/\mathtt{y}]) \models \mathrm{Post}(\mathtt{g},\vec{\mathtt{S}})$ by Lemma 6.3.4.

- Suppose the proof consists of an instance of the axiom scheme [fcall-a]:

$$\vdash \mathrm{Pre}(f,\vec{\mathtt{S}})\ \ \{\mathtt{y} \leftarrow \mathtt{f}(\vec{\mathtt{x}})\}\ \ \mathrm{Post}(f,\vec{\mathtt{S}}).$$

Suppose that $(C,\eta_C) \models \mathrm{Pre}(f,\vec{\mathtt{S}})$. By hypothesis, f satisfies its specification with respect to $SPEC$. So by definition for all possible results $r \in \mathcal{M}[\![f(\vec{\mathtt{x}})]\!](C,\eta_C)$, $r \neq \perp$ and $(C,\eta_C[r/\mathtt{y}]) \models \mathrm{Post}(f,\vec{\mathtt{S}})$.

For the inductive step, suppose that the result holds for all proofs of length less than n. Consider a proof of length $n > 1$. The last step of the proof must be either an axiom or the conclusion of an inference rule. The axioms were covered above, so it remains to deal with each of the rules of inference. Since all the conclusions of the rules of inference have a similar form the following conventions are established here to avoid repetition in each case. Let the nominal type of the expression (E) be $\mathtt{T}$. Let C be a $SPEC$-algebra and η_C an appropriate environment.

- Suppose the last step is the conclusion of the rule [mp-b]:

$$\vdash P\ \left\{\mathtt{y} \leftarrow \mathtt{g}(\vec{E})\right\}\ Q.$$

The hypothesis of this rule

$$\vdash P\ \left\{\mathtt{y} \leftarrow \mathtt{(fun\ (}\vec{\mathtt{x}}:\vec{\mathtt{S}}\mathtt{)\ g(}\vec{\mathtt{x}}\mathtt{))\ (}\vec{E}\mathtt{)}\right\}\ Q$$

must therefore appear in an earlier step. Since by definition of NOAL,

$$\mathcal{M}[\![\mathtt{g}(\vec{E})]\!](C,\eta_C) \quad = \quad \mathcal{M}[\![\mathtt{(fun\ (}\vec{\mathtt{x}}:\vec{\mathtt{S}}\mathtt{)\ g(}\vec{\mathtt{x}}\mathtt{))\ (}\vec{E}\mathtt{)}]\!](C,\eta_C) \tag{6.70}$$

the result follows.

- Suppose the last step is the conclusion of the rule [fcall-b]. Then the claim follows as for the rule [mp-b].

- The proof for [comb] is complex, so the following overview may serve as a guide to the details. The idea is that each possible result q_i of the actual argument expression E_i satisfies (by the inductive hypothesis) the condition $(R_i[\mathtt{v}_i/\mathtt{x}_i])[\vec{\mathtt{x}}/\vec{\mathtt{z}}]$. The renamings are shifted into the environment, so the assertions R_i characterize the formals of the function abstract. Then the inductive hypothesis is used to show that the possible results of the body E_0 satisfy the renamed post-condition $Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}]$. These renamings are also shifted into the environment, and then the renamings in the environment are manipulated to express the desired result.

Suppose the last step is the conclusion of the rule [comb]:

$$\vdash P\ \left\{\mathtt{y} \leftarrow \begin{array}{c}\mathtt{(fun\ (}\vec{\mathtt{x}}:\vec{\mathtt{S}}\mathtt{)\ }E_0\mathtt{)}\\ (E_1,\ldots,E_n)\end{array}\right\}\ Q.$$

Suppose $(C,\eta_C) \models P$.

There must be some earlier step in the proof of the form

$$\vdash R_1\ \&\ \cdots\ \&\ R_n\ \ \{\mathtt{y} \leftarrow E_0\}\ \ Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \tag{6.71}$$

where the $\vec{\mathtt{z}}:\vec{\mathtt{S}}$ are not free in P or Q, and none of the $\mathtt{x}_i$ is $\mathtt{y}$. By the inductive hypothesis, the above Hoare-triple is valid; that is:

$$(SPEC,FSPEC) \models \begin{array}{l} R_1\ \&\ \cdots\ \&\ R_n \\ \{\mathtt{y} \leftarrow E_0\} \\ Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \end{array}. \tag{6.72}$$

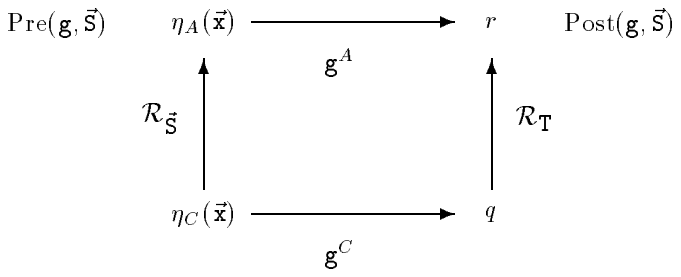

Figure 6.5: Soundness of the message passing axiom scheme.

For each i from 1 to n, there are earlier steps in the proof of the form:

$$\vdash P \ \{\mathtt{v}_i \leftarrow E_i\} \ (R_i[\mathtt{v}_i/\mathtt{x}_i])[\vec{\mathtt{x}}/\vec{\mathtt{z}}].$$

By the inductive hypothesis, this Hoare-triple is valid in (C, η_C). Since $(C, \eta_C) \models P$ by hypothesis, for all possible results $q_i \in \mathcal{M}[\![E_i]\!](C, \eta_C)$, $q_i \neq \bot$, and $(C, \eta_C[q_i/\mathtt{v}_i]) \models (R_i[\mathtt{v}_i/\mathtt{x}_i])[\vec{\mathtt{x}}/\vec{\mathtt{z}}]$. Let $\vec{q}$ be a tuple of the q_i that are possible results of the E_i. By the above $(C, \eta_C) \models (R_i[\mathtt{v}_i/\mathtt{x}_i])[\vec{\mathtt{x}}/\vec{\mathtt{z}}]$, and thus

$$\begin{array}{ll} & (C, \eta_C) \\ & \qquad (R_1[\mathtt{v}_1/\mathtt{x}_1])[\vec{\mathtt{x}}/\vec{\mathtt{z}}] \\ \models & \qquad \& \cdots \& \\ & \qquad (R_n[\mathtt{v}_n/\mathtt{x}_n])[\vec{\mathtt{x}}/\vec{\mathtt{z}}] \end{array} \tag{6.73}$$

By Lemma 6.3.2,

$$\begin{array}{l} (C, \eta_C[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{z}}]) \\ \quad \models \quad R_1[\mathtt{v}_1/\mathtt{x}_1] \,\&\, \cdots \,\&\, R_n[\mathtt{v}_n/\mathtt{x}_n] \end{array} \tag{6.74}$$

and applying Lemma 6.3.2 again:

$$\begin{array}{l} (C, (\eta_C[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{z}}])[\eta_C(\vec{\mathtt{v}})/\vec{\mathtt{x}}]) \\ \quad \models \quad R_1 \,\&\, \cdots \,\&\, R_n. \end{array} \tag{6.75}$$

Notice that these bindings place the values of the outer $\mathtt{x}_i$ variables in the fresh $\mathtt{z}_i$ and then the $\mathtt{v}_i$ in the $\mathtt{x}_i$. Let $\eta_C{}'$ be the environment defined by

$$\eta_C{}' \stackrel{\text{def}}{=} (\eta_C[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{z}}])[\eta_C(\vec{\mathtt{v}})/\vec{\mathtt{x}}]. \tag{6.76}$$

Since formula 6.72 holds, it follows that for all $r \in \mathcal{M}[\![E_0]\!](C, \eta_C{}')$, $r \neq \bot$ and $(C, \eta_C{}'[r/\mathtt{y}]) \models Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}]$.

Let $\eta_C{}''$ be defined by

$$\eta_C{}'' \stackrel{\text{def}}{=} \eta_C{}'[r/\mathtt{y}]. \tag{6.77}$$

Now by Lemma 6.3.2, the renamings on Q can be moved to the environment, hence:

$$(C, \eta_C{}''[\eta_C{}''(\vec{\mathtt{z}})/\vec{\mathtt{x}}]) \models Q \tag{6.78}$$

Since $\mathtt{y}$ is not the same as any of the $\mathtt{x}_i$, $\eta_C{}''(\vec{\mathtt{z}}) = \eta_C{}'(\vec{\mathtt{z}})$, and the binding of $\vec{\mathtt{z}}$ to $\eta'(\vec{\mathtt{x}})$ can be interchanged with the binding of $\mathtt{y}$ to r:

$$\eta_C{}''[\eta_C{}''(\vec{\mathtt{z}})/\vec{\mathtt{x}}] = (\eta_C{}'[\eta_C{}'(\vec{\mathtt{z}})/\vec{\mathtt{x}}])[r/\mathtt{y}]. \tag{6.79}$$

By construction $\eta_C{}'(\vec{\mathtt{z}}) = \eta_C(\vec{\mathtt{x}})$. Furthermore, if one extends an environment first with one binding for a variable and then with another binding for the variable, then the first extension no longer has any effect. Therefore,

$$\begin{array}{lll} & \eta_C{}'[\eta_C{}'(\vec{\mathtt{z}})/\vec{\mathtt{x}}] & \\ = & \eta_C{}'[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{x}}] & (6.80) \\ = & ((\eta_C[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{z}}])[\eta_C(\vec{\mathtt{v}})/\vec{\mathtt{x}}])[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{x}}] & (6.81) \\ = & (\eta_C[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{z}}])[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{x}}] & (6.82) \\ = & \eta_C[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{z}}]. & (6.83) \end{array}$$

So it follows from the above that

$$(C, (\eta_C[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{z}}])[r/\mathtt{y}]) \models Q. \tag{6.84}$$

Since the $\mathtt{z}_i$ are fresh, they do not appear free in Q, and thus

$$(C, \eta_C[r/\mathtt{y}]) \models Q. \tag{6.85}$$

By the definition of NOAL,

$$\begin{array}{ll} & \mathcal{M}[\![\mathtt{(fun\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{)}\ E_0\mathtt{)\ (}\vec{E}\mathtt{)}]\!](C, \eta_C) \\ = & \displaystyle\bigcup_{\vec{q} \in \mathcal{M}[\![\vec{E}]\!](C, \eta_C)} \mathcal{M}[\![E_0]\!](C, \eta_C[\vec{q}/\vec{\mathtt{x}}]). \end{array} \tag{6.86}$$

Hence each r above is in the denotation of the combination.

- Suppose the last step is the conclusion of the rule [if]:

  $$\vdash P \ \{\mathtt{y} \leftarrow \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{fi}\}\ Q.$$

  Suppose $(C, \eta_C) \models P$.

  There must be an earlier step in the proof of the form

  $$\vdash P \ \{\mathtt{v} \leftarrow E_1\} \ \mathrm{true}. \tag{6.87}$$

So by the inductive hypothesis, for all $r_1 \in \mathcal{M}[\![E_1]\!](C,\eta_C)$, $r_1 \neq \bot$.

Since E_1 has nominal type `Bool` and no other types are related to `Bool` by $\leq$, each r_1 must have type `Bool`.

There must be an earlier step in the proof of the form

$$\vdash P \,\&\, R_1 \ \{\mathtt{v} \leftarrow E_1\} \ \mathtt{v} = \text{true}. \tag{6.88}$$

By the inductive hypothesis, if $(C,\eta_C) \models P \,\&\, R_1$, then $\mathcal{M}[\![E_1]\!](C,\eta_C) = \{true\}$. Furthermore, there must be an earlier step in the proof of the form

$$\vdash P \,\&\, R_1 \ \{\mathtt{y} \leftarrow E_2\} \ Q. \tag{6.89}$$

So if $(C,\eta_C) \models P \,\&\, R_1$, then for all $r_2 \in \mathcal{M}[\![E_2]\!](C,\eta_C)$, $r_2 \neq \bot$ and $(C,\eta_C[r_2/\mathtt{y}]) \models Q$.

There must also be earlier steps in the proof of the form

$$\vdash \quad P \,\&\, R_2 \ \{\mathtt{v} \leftarrow E_1\} \ \mathtt{v} = \text{false} \tag{6.90}$$
$$\vdash \quad P \,\&\, R_2 \ \{\mathtt{y} \leftarrow E_3\} \ Q. \tag{6.91}$$

As above, if $(C,\eta_C) \models P \,\&\, R_2$, then for all $r_3 \in \mathcal{M}[\![E_3]\!](C,\eta_C)$, $r_3 \neq \bot$ and $(C,\eta_C[r_3/\mathtt{y}]) \models Q$.

There must also be earlier steps in the proof of the form

$$\vdash \quad P \,\&\, R_3 \ \{\mathtt{y} \leftarrow E_2\} \ Q \tag{6.92}$$
$$\vdash \quad P \,\&\, R_3 \ \{\mathtt{y} \leftarrow E_3\} \ Q. \tag{6.93}$$

So if $(C,\eta_C) \models P \,\&\, R_3$, then for all

$$r_{23} \in \mathcal{M}[\![E_2]\!](C,\eta_C) \cup \mathcal{M}[\![E_3]\!](C,\eta_C),$$

$r_{23} \neq \bot$ and $(C,\eta_C[r_{23}/\mathtt{y}]) \models Q$.

Finally, there must be an earlier step in the proof of the form

$$SPEC \vdash (R_1|R_2|R_3) = \text{true}. \tag{6.94}$$

Since $(C,\eta_C) \models P$, it follows that for some i from 1 to 3, $(C,\eta_C) \models P \& R_i$. Let r be a possible result of the `if` expression in (C,η_C); that is suppose that

$$r \in \bigcup_{r_1 \in \mathcal{M}[\![E_1]\!](C,\eta_C)} \begin{cases} \mathcal{M}[\![E_2]\!](C,\eta_C) \\ \qquad \text{if } r_1 = true \\ \mathcal{M}[\![E_3]\!](C,\eta_C) \\ \qquad \text{if } r_1 = false \\ \{\bot\} \\ \qquad \text{otherwise} \end{cases} \tag{6.95}$$

Then $r \neq \bot$, because by the above, the only possible results of E_1 are either *true* or *false*. Furthermore, for each i such that $\overline{\eta_C}[\![P \,\&\, R_i]\!] = true$, $r \neq \bot$ and $(C,\eta_C[r/\mathtt{y}]) \models Q$ by the above.

- Suppose the last step is the conclusion of the rule [erratic]:

$$\vdash P \ \{\mathtt{y} \leftarrow E_1 \,[\!]\, E_2\} \ Q.$$

Suppose $(C,\eta_C) \models P$. There must be earlier steps in the proof of the form:

$$\vdash \quad P \ \{\mathtt{y} \leftarrow E_1\} \ Q \tag{6.96}$$
$$\vdash \quad P \ \{\mathtt{y} \leftarrow E_2\} \ Q. \tag{6.97}$$

Each possible result of $E_1 \,[\!]\, E_2$ is a possible result of either E_1 or E_2. By the inductive hypothesis for each possible result r of either E_1 or E_2 in (C,η_C), $r \neq \bot$ and $(C,\eta_C[r/\mathtt{y}]) \models Q$.

- Suppose the last step is the conclusion of the rule [angelic]. Then the result follows as for the rule [erratic], since each possible result of $E_1 \bigtriangledown E_2$ is a possible result of either E_1 or E_2.

- Suppose the last step is the conclusion of the rule [isDef]:

$$\vdash P \ \{\mathtt{y} \leftarrow \mathtt{isDef?(}E\mathtt{)}\} \ \mathtt{y} = \text{true}.$$

Suppose $(C,\eta_C) \models P$. There must be an earlier step in the proof of the form

$$\vdash P \ \{\mathtt{y} \leftarrow E\} \ \text{true}.$$

By the inductive hypothesis, for all $r \in \mathcal{M}[\![E]\!](C,\eta_C)$, $r \neq \bot$. Therefore, by definition of NOAL, $\mathcal{M}[\![\mathtt{isDef?(}E\mathtt{)}]\!](C,\eta_C) = \{true\}$. Furthermore, since *true* is the only possible result it only remains to show that $(C,\eta_C[true/\mathtt{y}]) \models \mathtt{y} = \text{true}$, which is trivially true.

- Suppose the last step is the conclusion of the rule [conseq]:

$$\vdash P \ \{\mathtt{y} \leftarrow E\} \ Q.$$

Suppose $(C,\eta_C) \models P$.

There must be steps in the proof of the form $SPEC \vdash P \Rightarrow P_1$ and $SPEC \vdash Q_1 \Rightarrow Q$. In general, P_1 and Q_1 may have more free identifiers than P and Q. For example, the formula "true $\Rightarrow_{\vec{\mathtt{z}}:\vec{\mathtt{S}}}$ `i = i`" and its converse are both valid. Let $\vec{\mathtt{z}} : \vec{\mathtt{S}}$ be a tuple of all the free identifiers of P_1 and Q_1 except for the result identifier $\mathtt{y} : \mathtt{T}$ that are not in X (i.e., that are not in the domain of η_C). Let $\vec{q} \in \vec{\mathtt{S}}^C$ be a tuple of proper elements. Since $SPEC \vdash P \Rightarrow P_1$ and since P and P_1 are subtype-constraining, by Lemma 6.3.5, $(C,\eta_C[\vec{q}/\vec{\mathtt{z}}]) \models P \Rightarrow P_1$. Since $(C,\eta_C[\vec{q}/\vec{\mathtt{z}}]) \models P$, it follows that

$$(C,\eta_C[\vec{q}/\vec{\mathtt{z}}]) \models P_1. \tag{6.98}$$

There must also be a step in the proof of the form $\vdash P_1 \ \{\mathtt{y} \leftarrow E\} \ Q_1$. By the inductive hypothesis and the above, for all $r \in \mathcal{M}[\![E]\!](C,\eta_C[\vec{q}/\vec{\mathtt{z}}])$, $r \neq \bot$, and

$$(C,(\eta_C[\vec{q}/\vec{\mathtt{z}}])[r/\mathtt{y}]) \models Q_1. \tag{6.99}$$

Since Q and Q_1 are subtype-constraining, by Lemma 6.3.5, $(C,(\eta_C[\vec{q}/\vec{\mathtt{z}}])[r/\mathtt{y}]) \models Q_1 \Rightarrow Q$ and thus

$$(C,(\eta_C[\vec{q}/\vec{\mathtt{z}}])[r/\mathtt{y}]) \models Q. \tag{6.100}$$

Since the $\mathtt{z}_i$ are not free in Q,

$$(C,\eta_C[r/\mathtt{y}]) \models Q. \tag{6.101}$$

- Suppose the last step is the conclusion of the rule [equal]:

$$\vdash P\ \{\mathtt{y} \leftarrow E\}\ M[\mathtt{y}/\mathtt{z}] = M[N/\mathtt{z}].$$

where $\mathtt{y}, \mathtt{z} : \mathtt{T}$. Suppose $(C, \eta_C) \models P$. There must be an earlier step in the proof of the form $\vdash P\ \{\mathtt{y} \leftarrow E\}\ \mathtt{y} = N$. By the inductive hypothesis, for all $r \in \mathcal{M}[\![E]\!](C, \eta_C)$, $r \neq \perp$ and

$$(C, \eta_C[r/\mathtt{y}]) \models \mathtt{y} = N. \tag{6.102}$$

Therefore for all such r, $r = \overline{\eta_C[r/\mathtt{y}]}[\![N]\!]$, and thus

$$(C, \eta_C[r/\mathtt{y}]) \models M[\mathtt{y}/\mathtt{z}] = M[N/\mathtt{z}]. \tag{6.103}$$

- Suppose the last step is the conclusion of the rule [carry]:

$$\vdash P\ \{\mathtt{y} \leftarrow E\}\ P \,\&\, Q.$$

Suppose $(C, \eta_C) \models P$. There must be an earlier step in the proof of the form

$$\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q.$$

By the inductive hypothesis, for all $r \in \mathcal{M}[\![E]\!](C, \eta_C)$, $r \neq \perp$ and

$$(C, \eta_C[r/\mathtt{y}]) \models Q. \tag{6.104}$$

Since $(C, \eta_C) \models P$, and $\mathtt{y}$ is not free in P:

$$(C, \eta_C[r/\mathtt{y}]) \models P. \tag{6.105}$$

Therefore,

$$(C, \eta_C[r/\mathtt{y}]) \models P \,\&\, Q. \tag{6.106}$$

- Suppose the last step is the conclusion of the rule [rename]:

$$\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q.$$

Since the identifiers $\vec{\mathtt{z}}$ are fresh, the possible results of $E[\vec{\mathtt{z}}/\vec{\mathtt{x}}]$ in $\eta_C[\eta_C(\vec{\mathtt{x}})/\vec{\mathtt{z}}]$ are the same as the possible results of E in (C, η_C).

∎

In the above lemma, it was assumed that the NOAL functions from the surrounding program satisfied their specifications. By making such an assumption, the above result can be used to show the soundness of the Hoare logic for proving the partial correctness of NOAL programs and recursively defined NOAL functions. Since the treatment of verification of recursive functions is standard, the partial correctness of recursive functions is not formally proved below. Such a proof would proceed by fixedpoint induction, but since there are three fixedpoint constructions used to define the semantics of recursively defined NOAL functions (due to the use of erratic and angelic choice, see Appendix C), the proof is complex and outside the scope of this report.

The following corollary is the soundness result for program verification. It is a trivial consequence of the above lemma and Lemma 5.3.4, which says that the possible results of a type-safe NOAL program must be instances of subtypes of the program's nominal type. This connection to the type system of NOAL is explored further below.

Corollary 6.3.7. Let p be a program specification with pre-condition R and post-condition Q and nominal result type `S`. Let $(SPEC, FSPEC)$ be a pair of type and function specification sets. Let $\leq$ be the presumed subtype relation of $SIG(SPEC)$. Let P be the NOAL program $\vec{F}$ `; program (`$\vec{\mathtt{x}}$ `:` $\vec{\mathtt{T}}$`):S =`E where $\vec{F}$ is a system of mutually recursive NOAL functions whose names and nominal signatures match $SIG(FSPEC)$.

Suppose that $\leq$ is a subtype relation on the types of $SPEC$ and the denotation of each function in $\vec{F}$ satisfies its specification with respect to $SPEC$. If $(SPEC, FSPEC) \vdash R\ \{\mathtt{y} \leftarrow E\}\ Q$, then the program P satisfies the specification p with respect to $SPEC$.

∎

6.4 Modularity

The soundness results of the previous section do not completely vindicate the claim that the Hoare logic allows modular reasoning. The soundness result shows that one can, for a given set of type specifications, reason about a function or program using nominal type information without explicitly considering subtypes. Yet modularity demands that such verifications still be valid when new subtypes are added to a program. The precise notion of extension is given in the following definition.

Definition 6.4.1 (extends).
Let $SPEC_1$ and $SPEC_2$ be sets of type specifications. The set $SPEC_2$ *extends* $SPEC_1$ if and only if the type specifications $SPEC_1$ are included in $SPEC_2$ and $SIG(SPEC_1)$ is a subsignature of $SIG(SPEC_2)$

The requirement that the original signature be a subsignature of the extension's guarantees that no new subtype relationships are added between the original types, and that the nominal result types of existing expressions must be subtypes of their original types.

The following lemma and its corollary shows that the verification of expressions is modular. It states that a verification using an smaller set of specifications necessarily is a verification using an extended set of specifications. In other words, the extended set's theory includes the smaller's theory.

Lemma 6.4.2. Let $SPEC_1$ and $SPEC_2$ be sets of type specifications. Let $FSPEC$ be a set of function specifications whose base specification set is contained in $SPEC_1$.

Suppose that the set of type specifications $SPEC_2$ extends the set $SPEC_1$. Then for all Hoare-triples for $(SPEC_1, FSPEC)$, if

$$(SPEC_1, FSPEC) \vdash P\ \{\mathtt{y} \leftarrow E\}\ Q,$$

then

$$(SPEC_2, FSPEC) \vdash P\ \{\mathtt{y} \leftarrow E\}\ Q.$$

Proof: Suppose that

$$(SPEC_1, FSPEC) \vdash P\ \{\mathtt{y} \leftarrow E\}\ Q. \tag{6.107}$$

Since $SPEC_2$ extends $SPEC_1$, each axiom of the pair $(SPEC_1, FSPEC)$ is an axiom of $(SPEC_2, FSPEC)$.

Since the signature $SIG(SPEC_1)$ is a subsignature of $SIG(SPEC_2)$, by Lemma 5.3.1, the nominal type of each NOAL expression E with respect to $SIG(SPEC_1)$ and $SIG(FSPEC)$ is a supertype of the nominal type of the expression E's nominal type with respect to $SIG(SPEC_2)$ and $SIG(FSPEC)$. So each Hoare-triple for $(SPEC_1, FSPEC)$ is a Hoare-triple for $(SPEC_2, FSPEC)$.

It must also be checked that the type constraints of the Hoare-logic's inference rules are satisfied. The actual arguments $\vec{E}$ in the inference rules [mp-b] and [fcall-b] have types $\vec{\sigma}$ with respect to $SIG(SPEC_1)$ and $SIG(FSPEC)$, and $\vec{\sigma} \leq_1 \vec{\mathtt{S}}$, where $\leq_1$ is the presumed subtype relation of $SIG(SPEC_1)$. By Lemma 5.3.1, the nominal types of the actual arguments $\vec{E}$ must be some $\vec{\tau} \leq_2 \vec{\sigma}$. Since $\leq_1 \subseteq \leq_2$, $\vec{\tau} \leq_2 \vec{\mathtt{S}}$. The type constraints on the inference rules [equal] and [rename] only ensure that the nominal sorts of certain identifiers are the same. Therefore the proof for the triple is a proof in $(SPEC_2, FSPEC)$. ∎

The following corollary states that an expression verification done with a smaller specification is valid for an extended specification, if the presumed subtype relation on the extended specification satisfies the semantic constraints for subtype relations. The proof is direct from the above lemma and Lemma 6.3.6.

Corollary 6.4.3. Let $SPEC_1$ and $SPEC_2$ be sets of type specifications. Let $FSPEC$ be a set of function specifications whose base specification set is contained in $SPEC_1$.

Suppose that $SPEC_2$ extends $SPEC_1$. Suppose that the presumed subtype relation $\leq_2$ of $SIG(SPEC_2)$ is a subtype relation on the types of $SPEC_2$. Then for all Hoare-triples for $(SPEC_1, FSPEC)$, if

$$(SPEC_1, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E\} \ Q,$$

then

$$(SPEC_2, FSPEC) \models P \ \{\mathtt{y} \leftarrow E\} \ Q.$$

∎

The story for modularity is not, however, as simple as the above corollary would indicate. The complication is that the implementations of NOAL functions are verified using the smaller type specification set but not reverified using the expanded set of type specifications. Since the verification of recursively defined NOAL functions using the Hoare logic only shows partial correctness, knowing that proof of partial correctness using the smaller specification set gives a proof of partial correctness for the expanded specification set is not enough to satisfy the conditions of Corollary 6.3.7. To avoid redoing the proof of termination of recursively defined NOAL functions one needs to know that if a NOAL function satisfies its specification with respect to the smaller set of type specifications, then it satisfies its specification with respect to an expanded set of type specifications.

The following lemma asserts that such problems do not occur for NOAL functions, provided that the new subtype relation satisfies the necessary semantic constraints. The proof is the source of the restriction that function specifications may only use subtype-constraining assertions.

Lemma 6.4.4. Let $SPEC_1$ and $SPEC_2$ be sets of type specifications. Let f be a function specification whose base specification set is contained in $SPEC_1$. Let f be the denotation of a NOAL function definition for f.

Suppose that $SPEC_2$ extends $SPEC_1$. Suppose that the presumed subtype relation $\leq_2$ of $SIG(SPEC_2)$ is a subtype relation on the types of $SPEC_2$. If f satisfies the specification f with respect to $SPEC_1$, then f satisfies the specification f with respect to $SPEC_2$.

Proof: Suppose that f satisfies the specification f with respect to $SPEC_1$. Let C be a $SPEC_2$-algebra. Let X be a set of identifiers that includes the formal arguments from the specification of f. Let $\mathtt{S}$ be the nominal result type of f. Let $\eta_C : X \rightarrow |C|$ be a proper $SIG(SPEC_2)$-environment. Let R be the pre-condition of f, and let Q be its post-condition and $\mathtt{v}$ the formal result identifier. Suppose that

$$(C, \eta_C) \models R. \tag{6.108}$$

Let $q \in f(C)(\eta_C(\vec{\mathtt{x}}))$ be a possible result of f.

Since $SPEC_2$ extends $SPEC_1$ and since η_C is a $SIG(SPEC_2)$-environment, it must be that the nominal type of each $\mathtt{x}_i$ is some $\mathtt{T}_i$ and each $\eta_C(\mathtt{x}_i)$ has a type that U_i, such that $\mathtt{U}_i \leq \mathtt{T}_i$. Since $\leq_2$ is a subtype relation, there must be some $SPEC_2$-algebra A and a $SIG(SPEC_2)$-simulation relation, $\mathcal{R}$, from C to A. Let η_A be a nominal environment such that $\eta_C \, \mathcal{R} \, \eta_A$; such an environment can be constructed by the coercion property of simulation relations. Let A' be the $SIG(SPEC_1)$-reduct of A. Since η_A is nominal and the base specification of f is contained in $SPEC_1$, the nominal types of the formals of the $\mathtt{x}_i$ must be types in $SIG(SPEC_1)$, hence for each i, $\eta_A(\mathtt{x}_i) \in A'$.

Since f is the denotation of a NOAL function definition and $\mathcal{R}$ is a simulation relation from C to A, by Lemma 7.2.2, there is some possible result $r \in f(A)(\eta_A(\vec{\mathtt{x}}))$, such that $q \, \mathcal{R}_{\mathtt{S}} \, r$ (where $\mathtt{S}$ is the nominal result type of f). Since R is subtype-constraining, by Lemma 6.3.4,

$$(A, \eta_A) \models R. \tag{6.109}$$

Since the function f satisfies its specification with respect to $SPEC_1$, and since $\eta_A(\vec{\mathtt{x}})$ is in the $SPEC_1$-algebra A',

$$(A, \eta_A[r/\mathtt{v}]) \models Q. \tag{6.110}$$

Since Q is subtype-constraining and $\eta_C[q/\mathtt{v}] \mathcal{R} \, \eta_A[r/\mathtt{v}]$, by Lemma 6.3.4,

$$(C, \eta_C[q/\mathtt{v}]) \models Q. \tag{6.111}$$

∎

The following corollary gives the modularity result for program verification. It may seem that the corollary discusses adding new types to a program and then the new types are never used, because the program is unchanged. However, a NOAL program may take arguments of any type, and so it may have an argument whose nominal type is a supertype of a newly added type. Hence the old program may be passed objects of the new type, which is precisely what programmers are concerned with. One should perhaps think of a NOAL program in this context as an abstraction of the part of a "real" program that processes objects after they have been constructed from the "real" program's input.

Corollary 6.4.5. Let p be a program specification, pre-condition R and post-condition Q and nominal result type `S`. Let $(SPEC_1, FSPEC)$ be a pair of type and function specification sets. Let $SPEC_2$ be an extension of $SPEC_1$. Let $\leq_2$ be the presumed subtype relation of $SIG(SPEC_2)$. Let P be the NOAL program $\vec{F}$; `program (`$\vec{\mathtt{x}}$ `:` $\vec{\mathtt{T}}$`):S =`E where $\vec{F}$ is a system of mutually recursive NOAL functions whose names and nominal signatures match $SIG(FSPEC)$.

Suppose that $\leq_2$ is a subtype relation on the types of $SPEC_2$ and the denotation of each function in $\vec{F}$ satisfies its specification with respect to $SPEC_1$. If $(SPEC_1, FSPEC) \vdash R\ \{\mathtt{y} \leftarrow E\}\ Q$, then the program P satisfies the specification p with respect to $SPEC_2$.

Proof: By the previous lemma, each the denotation of each function in $\vec{F}$ satisfies its specification with respect to $SPEC_2$. Suppose that

$$(SPEC_1, FSPEC) \vdash R\ \{\mathtt{y} \leftarrow E\}\ Q. \qquad (6.112)$$

By Lemma 6.4.2, it follows that

$$(SPEC_2, FSPEC) \vdash R\ \{\mathtt{y} \leftarrow E\}\ Q. \qquad (6.113)$$

∎

6.5 How a Type System can Aid Verification

The aid that a type-checker can give a verifier is discussed in this section.

6.5.1 Obedience in NOAL

For soundness of the verification technique for NOAL programs, the possible results of each expression must be instances of a subtype of the expression's nominal type. This property is called *obedience*. Fundamentally, obedience is necessary to prevent functions from being invoked outside their domains. Thus obedience is crucial for soundness and was used in the soundness theorems above.

Instead of checking obedience with the Hoare logic, it is convenient to separate type checking from the rest of the verification problem. Separating type checking from verification allows the logic to be simpler than it would be otherwise. Furthermore, type checking can be mechanical, as in Trellis-Owl [SCB+86].

The NOAL type system can ensure obedience of type-safe expressions over a specification $SPEC$ if the conditions on signatures are met. See Section 5.3.

6.5.2 Verification in Trellis/Owl

It is easiest to use the Hoare-style verification techniques described above in a statically typed object-oriented programming language, such as Trellis/Owl [SCB+86]. The Trellis/Owl type system was the inspiration for the NOAL type system, since it is static and based on nominal signatures and a declared subtype relation.

Trellis/Owl limits presumed subtype relations to be partial orders, that is reflexive, transitive, and antisymmetric relations on types. Although reflexive and transitive relations are necessary for type-checking and verification, antisymmetry is not. Trellis/Owl requires that a presumed subtype relation be antisymmetric, because the implementation of each presumed subtype is also a subclass, and cyclic inheritance relationships are nonsensical. However, symmetric subtype relationships are useful. For example, consider types `HashTable` and `BTree`. One can specify these types so instances of these types obey a common protocol for inserting and finding elements and so that each is a subtype of the other, although they can have different class operations.

The Trellis/Owl type system supports program verification by ensuring obedience to the declared subtype relation. If the declared subtype relation satisfies the semantic constraints described in Chapter 4, then the style of reasoning described above should be useful for program verification in Trellis/Owl.

6.5.3 Verification in Emerald

Unlike Trellis/Owl the designers of Emerald [BHJL86] have made the mistake of inferring subtype relationships for abstract types from syntactic interfaces. Unfortunately, it is easy to specify types so that the binary relation that Emerald infers is not a subtype relation (i.e., the inferred relation does not satisfy the semantic constraints on subtype relations). Therefore to ensure that every environment obeys a subtype relation in an Emerald program, one has to duplicate work that the type checker could have done.

6.5.4 Verification in Smalltalk-80

Many object-oriented languages, such as Smalltalk-80 and CLOS are not statically type-checked but are type-checked dynamically. In Smalltalk-80, type information is not checked during assignments, but only on message sends. To support data abstraction, Smalltalk-80 ensures that each object is manipulated only by the instance operations defined by its class (including those inherited from superclasses). Therefore, when an instance operation named `g` is invoked on an object q, the class that implements q must define operation `g`; if it does not, an error occurs and is reported to the user.

To use the above reasoning techniques on Smalltalk-80 programs, one needs a notion of nominal type and some way to ensure obedience to a subtype relation.

To supply Smalltalk-80 programs with a notion of nominal type, one can annotate one's programs with this information. The Smalltalk-80 programs in Goldberg and Robson's book [GR83] already follow a convention of putting type information into variable names to aid understanding.

There are two ways to force expressions to obey a subtype relation: dynamic or static checking. Notice that one cannot rely on the dynamic type-checking of Smalltalk-80 to ensure obedience, because Smalltalk-80 only checks that an instance operation invoked on an object q is defined by q's class.

Dynamic checking could use the type information available at run-time in Smalltalk-80 programs. One would place code in all operations to check that all the operation's arguments have a type that is a subtype of their nominal type. (Smalltalk-80 itself checks the first or "controlling" argument, so no checking on the

first argument is needed.) A Smalltalk-80 program with such dynamic type checks is said to be obedient if these checks never detect an instance of some type other than a subtype. Of course, there is no general algorithm for deciding when a Smalltalk-80 program is obedient.

Another way to ensure obedience would be to do static type-checking using the nominal type information added to programs as annotations. It should be easy to adapt the NOAL type system to Smalltalk-80, which would allow one to do some type checking by hand, or to write a tool that used program annotations to do static type checking.

Chapter 7

Observability

In this chapter the behavioral properties of subtype relations and simulation are discussed. The results can be thought of as another justification for the definition of subtype relations given in Chapter 4, since it is shown that subtyping prevents surprising behavior. Thus the definition of subtype relations agrees with the intuition that each instance of a subtype can be manipulated as if it were an instance of the supertype.

The main tools for this investigation are the notions of weak subtype relations and imitation. Weak subtypes are like subtypes, and imitation is like simulation. However, weak subtyping and imitation vary with respect to a set of observations. For example, q might imitate r with respect to programs that only observe them by using the message `size`, yet q might not imitate r with respect to all NOAL programs. This context dependence allows one to investigate notions of subtyping appropriate for particular languages or language subsets [Lea89]. It also shows how type checking is crucial for preventing surprising behavior, since it prevents programs from sending messages to objects to which a subtype instance might react differently than a supertype instance. (The subtype instance might do something useful, while the supertype might just give an error.)

The results of this section are largely independent of the specification language and the programming language NOAL. NOAL programs are used as a notation for observations, however.

In the rest of this chapter imitation is described first, and then the relation of simulation and imitation is discussed. Then weak subtype relations are described and it is shown that subtyping implies weak subtyping. The final discussion concerns testing and a comparison of the two notions of subtyping.

7.1 Observations and Imitation

Intuitively, an observation is a program. Indeed, the denotation of a NOAL program is formally an observation. A set of observations is thus like a programming language or a subset of a programming language. For example, the set of observations defined by all type-safe NOAL programs is used often in what follows.

A behavior of an object is a result from a program that takes the object as an argument.

The set of possible results of an observation are what might be seen by a superhuman tester who runs the program over and over again. This superhuman tester is able to "observe" all possible results, even when a possible result is nontermination ($\perp$) or when the set of possible results is infinite. Real testers are have limitations that are reflected in the ways that sets of possible results are compared.

Definition 7.1.1 (Σ-observation). Let Σ be a signature, and X is a set of typed identifiers. A *Σ-observation with free identifiers from X* is a mapping that takes a Σ-algebra A and a Σ-environment $\eta : Y \to |A|$ such that $X \subseteq Y$, and returns a set of possible results from A such that each possible result has a visible type.

For example,

$$\lambda(A, \eta) \,.\, \mathcal{M}[\![\texttt{elem(x,2)}]\!](A, \eta)$$

is a $SIG(II)$-observation with free identifiers `x : IntSet` that tests whether 2 is in `x` using the operation `elem`. This observation is the denotation of the following NOAL program.

```
program (x:IntSet):Bool = elem(x,2)
```

The environment of the algebra-environment pair that an observation takes as an argument provides a way to access the objects to be observed.

The notion of imitation is the behavioral analog of simulation (which is a purely algebraic notion). If q imitates r, then q's behaviors should not be distinguishable from r's; that is, the behavior of q should not surprise someone[1] that expected to be observing r. The concept of "imitation" defined below compares the behaviors of algebra-environment pairs. This allows one to discuss the behavior of several objects at a time. In addition, algebra-environment pairs come equipped with type assumptions about objects, since objects are accessible from the environment only through typed identifiers.

For deterministic algebras, a satisfactory notion of imitation is observable equivalence. The algebra-environment pair (C, η_C) *is observably equivalent to* (A, η_A) *with respect to a set of observations OBS* if and only if for all observations $P \in OBS$ with free identifiers from some subset of X, $P(C, \eta_C) = P(A, \eta_A)$.

However, observable equivalence is too strong for nondeterministic algebras. For nondeterministic algebras, one algebra-environment pair should be allowed to imitate another, even if the first is more deterministic. This corresponds to a limitation of human observers: one cannot predict which possible results will be exhibited by a particular run of a program, hence the lack of some possible results cannot be definitely established by testing. Thus specifications limit the set of possible results, but do not need to completely

[1] A person, not a superhuman.

determine the exact set of possible results of a nondeterministic program. For example, if one specifies that a procedure `g` may return any even number, then one should be satisfied with an implementation of `g` that can only return 4 or 8, it is not necessary for an implementation of `g` to also be able to return 16.

Definition 7.1.2 (imitates). Let Σ be a signature. Let X be a set of typed identifiers. Let OBS be a set of Σ-observations with free identifiers from X. Let C and A be Σ-algebras. Let $\eta_C : X \to |C|$ and $\eta_A : X \to |A|$ be Σ-environments. Then the pair (C, η_C) *imitates* (A, η_A) *with respect to* OBS if and only if for all observations $P \in OBS$ with free identifiers from some subset of X, $P(C, \eta_C) \subseteq P(A, \eta_A)$.

For example, let P be the following observation

$$P = \mathcal{M}[\![\texttt{program (x:IntSet):Int = choose(x)}]\!]. \tag{7.1}$$

Consider II-algebras C and A and environments $\eta_C : \{\texttt{x} : \texttt{IntSet}\} \to C$ and $\eta_A : \{\texttt{x} : \texttt{IntSet}\} \to A$ such that

$$P(C, \eta_C) = \{1, 2\} \tag{7.2}$$
$$P(A, \eta_A) = \{1, 2, 3, 4\}. \tag{7.3}$$

Then (C, η_C) imitates (A, η_A) with respect to $\{P\}$, but not vice versa.

When the set of observations is fixed, we simply say that (C, η_C) imitates (A, η_A).

In general, the imitates relation with respect to a fixed set of observations is not symmetric, as the example above shows. It is reflexive and transitive, however.

Lemma 7.1.3. Let Σ be a signature. Let OBS be a set of Σ-observations.

Then the imitates relation with respect to OBS is a preorder. ∎

On the other hand, for deterministic algebras, the imitates relation is symmetric and the same as observable equivalence.

Whenever one algebra-environment pair does *not* imitate another with respect to a set of observations, then there is some observation in that set that shows a difference. However, if one thinks of running the program that defines an observation in a real implementation, one might have to wait forever to "see" the difference, because the difference may be that the program fails to halt or that it might produce some result nondeterministically. In the example above, (A, η_A) does not imitate (C, η_C) with respect to the observation P that sends the message `choose`, because 3 is a possible result from (A, η_A). However, in a real implementation there is no guarantee that the result "3" will be produced in (A, η_A) at any time.

The following facts about the how the imitates relation depends on sets of observations are useful for comparing different programming languages. They are similar to facts about observable equivalence studied by others [ST85, Facts 2–3].

The first lemma below says that the imitates relation with respect to a larger set of observations is a subset of the imitates relation with respect to smaller sets of observations. As an extreme example, the imitates relation with respect to the empty set of observations relates all algebra-environment pairs. In general, adding observations may allow one to observe more differences.

Lemma 7.1.4. Let Σ be a signature. Let OBS and OBS' be sets of Σ-observations.

If $OBS \supseteq OBS'$ and (C, η_C) imitates (A, η_A) with respect to OBS, then (C, η_C) imitates (A, η_A) with respect to OBS'. ∎

The following says that the imitates relation with respect to a set of observations OBS is the intersection of the imitates relations with respect to all subsets of OBS.

Lemma 7.1.5. Let Σ be a signature. Let $OBS = \bigcup_{i \in I} OBS_i$ be a set of Σ-observations.

If for each $i \in I$, (C, η_C) imitates (A, η_A) with respect to OBS_i, then (C, η_C) imitates (A, η_A) with respect to OBS. ∎

7.2 Simulation as a Criteria for Imitation

The goal of this section is to show that simulation is stronger than imitation with respect to NOAL programs. This result is a justification of the definition of simulation; that is, simulation is preserved by NOAL expressions and programs.

The following lemmas shows that simulation is preserved by NOAL programs, not just be single invocations of program operations. This property is analogous to the "fundamental theorem of logical relations" [Sta85].

The first lemma shows that simulation is preserved by all type-safe NOAL expressions. This is done in two steps. The first step assumes that the denotations of NOAL functions are related by a simulation relation (in a way described below). The second lemma, which states that the meaning of a function definition is appropriately related in the related algebras, is summarized in this subsection, and proved formally in Appendix C.

For a given algebra, the denotation of a NOAL function is a mapping from tuples of arguments to sets of possible results. Such mappings are related by analogy to the definition of logical relations [Sta85] [Mit86]. That is, if $\mathcal{R}$ is family of sorted relations, it is extended to the signatures of NOAL function identifiers as follows:

$$\mathcal{R}_{\vec{\tau} \to \sigma} \stackrel{\text{def}}{=} \{(f_1, f_2) \mid \vec{q}\,\mathcal{R}_{\vec{\tau}}\,\vec{r} \Rightarrow f_1(\vec{q})\,\mathcal{R}_\sigma\,f_2(\vec{r})\}. \tag{7.4}$$

That is, for all f_1 and f_2, f_1 is related by $\mathcal{R}_{\vec{\tau} \to \sigma}$ to f_2 if and only if whenever $\vec{q}\,\mathcal{R}_{\vec{\tau}}\,\vec{r}$, then for every $q' \in f_1(\vec{q})$, there is some $r' \in f_2(\vec{r})$ such that $q'\,\mathcal{R}_\sigma\,r'$. Notice that this extension of $\mathcal{R}$ preserves the substitution property of simulation relations. Since operations are not first-class objects in NOAL, it is not necessary to show that this extension has all the properties of a simulation relation at each function signature.

To deal with NOAL expressions that have free function identifiers, environments are allowed to map typed

function identifiers to their denotations in the algebra that is the environment's range (i.e., to set-valued functions).

For brevity throughout the rest of this section, fix a signature Σ and Σ-algebras A and B. As usual Σ is such that:

$$\Sigma = \left(\begin{array}{l} SORTS, TYPES, V, \leq, \\ SFUNS, POPS, ResSort \end{array} \right).$$

Informally, the following lemma says that simulation is preserved by NOAL expressions if it is preserved by each recursively defined function.

Lemma 7.2.1. Let X be a set of typed identifiers and function identifiers. Let $\eta_1 : X \to |A|$ and $\eta_2 : X \to |B|$ be Σ-environments.

If $\mathcal{R}$ is a Σ-simulation relation between A and B and if $\eta_1 \, \mathcal{R} \, \eta_2$, then for all types $\mathtt{T}$ and for all NOAL expressions E of nominal type $\mathtt{T}$ whose free identifiers and function identifiers are a subset of X,

$$\mathcal{M}[\![E]\!](A, \eta_1) \, \mathcal{R}_{\mathtt{T}} \, \mathcal{M}[\![E]\!](B, \eta_2).$$

Proof: (by induction on the structure of expressions).

For the basis, suppose that the expression is either an identifier or `bottom[T]`. If the expression is an identifier, then the result follows from $\eta_1 \, \mathcal{R} \, \eta_2$. If the expression is `bottom[T]` for some type $\mathtt{T}$, then the result follows from the bistrictness of $\mathcal{R}_{\mathtt{T}}$.

For the inductive step, assume that if $\eta_1 \, \mathcal{R} \, \eta_2$, then the denotation of each subexpression of nominal type $\mathtt{T}$ in the environment η_1 is related by $\mathcal{R}_{\mathtt{T}}$ to the denotation of the same subexpression in the environment η_2. There are several cases (see Figures 5.1 and 5.2).

- Suppose the expression is $\mathtt{g}(\vec{E})$. Since this expression has a nominal type, by the type inference rules it must be that $\vec{E} : \vec{\sigma}$, and $ResSort(\mathtt{g}, \vec{\sigma}) = \mathtt{T}$. Let $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_1)$ be given. By the inductive hypothesis, there is some $\vec{r} \in \mathcal{M}[\![\vec{E}]\!](B, \eta_2)$ such that $\vec{q} \, \mathcal{R}_{\vec{\sigma}} \, \vec{r}$. Since $\vec{q} \, \mathcal{R}_{\vec{\sigma}} \, \vec{r}$ and $\mathcal{R}$ is a simulation relation, it follows that

  $$\mathtt{g}^A(\vec{q}) \, \mathcal{R}_{\mathtt{T}} \, \mathtt{g}^B(\vec{r}). \tag{7.5}$$

  Therefore, for each $q \in \mathcal{M}[\![\mathtt{g}(\vec{E})]\!](A, \eta_1)$ there is some $r \in \mathcal{M}[\![\mathtt{g}(\vec{E})]\!](B, \eta_2)$ such that $q \, \mathcal{R}_{\mathtt{T}} \, r$.

- Suppose the expression is $f(\vec{E})$ and f is a function identifier with nominal signature $\vec{\mathtt{S}} \to \mathtt{T}$. Since this expression has a nominal type, by the type inference rules it must be that $\vec{E}$ has nominal type $\vec{\sigma}$ and $\vec{\sigma} \leq \vec{\mathtt{S}}$. Let $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_1)$ be given. By the inductive hypothesis, there is some $\vec{r} \in \mathcal{M}[\![\vec{E}]\!](B, \eta_2)$ such that $\vec{q} \, \mathcal{R}_{\vec{\sigma}} \, \vec{r}$. Since $\vec{\sigma} \leq \vec{\mathtt{S}}$, by the coercion properties of a Σ-simulation relation, $\mathcal{R}_{\vec{\sigma}} \subseteq \mathcal{R}_{\vec{\mathtt{S}}}$, and so $\vec{q} \, \mathcal{R}_{\vec{\mathtt{S}}} \, \vec{r}$. Since $\eta_1 \, \mathcal{R} \, \eta_2$,

  $$\eta_1(f) \, \mathcal{R}_{\vec{\mathtt{S}} \to \mathtt{T}} \, \eta_2(f), \tag{7.6}$$

  and thus

  $$(\eta_1(f))(\vec{q}) \, \mathcal{R}_{\mathtt{T}} \, (\eta_2(f))(\vec{r}). \tag{7.7}$$

  So, by definition of NOAL, for every possible result $q \in \mathcal{M}[\![f(\vec{E})]\!](A, \eta_1)$ there is some $r \in \mathcal{M}[\![f(\vec{E})]\!](B, \eta_2)$ such that $q \, \mathcal{R}_{\mathtt{T}} \, r$.

- Suppose the expression is $(\mathtt{fun}(\vec{\mathtt{x}} : \vec{\mathtt{S}})E_0)(\vec{E})$ and that the nominal type of the entire expression is $\mathtt{T}$. Let $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_1)$ be given. By the inductive hypothesis, there is some $\vec{r} \in \mathcal{M}[\![\vec{E}]\!](B, \eta_2)$ such that $\vec{q} \, \mathcal{R}_{\vec{\sigma}} \, \vec{r}$, where $\vec{\sigma}$ is the nominal type of $\vec{E}$. Since the expression has a nominal type, by the type inference rules for NOAL it must be that $\vec{\sigma} \leq \vec{\mathtt{S}}$; thus $\vec{q} \mathcal{R}_{\vec{\mathtt{S}}} \vec{r}$. It follows that if one binds $\vec{\mathtt{x}}$ to $\vec{q}$ in η_1 and $\vec{\mathtt{x}}$ to $\vec{r}$ in η_2, then $(\eta_1[\vec{q}/\vec{\mathtt{x}}]) \, \mathcal{R} \, (\eta_2[\vec{r}/\vec{\mathtt{x}}])$; thus the result follows by the inductive hypothesis (applied to E_0).

- Suppose the expression is `if` E_1 `then` E_2 `else` E_3 `fi`. Since `Bool` is a visible type and $\mathcal{R}$ is V-identical, the possible results from E_1 in η_1 are a subset of those possible in η_2. Therefore the result follows from the inductive hypothesis applied to E_2 and E_3.

- Suppose the expression is $E_1 \sqcap E_2$. The possible results of this expression are the union of those from E_1 and E_2. Since the expression has a nominal type, there is a type $\mathtt{T}$ that is the least upper bound of the nominal types of E_1 and E_2. Let the nominal type of E_1 be $\mathtt{S}_1$ and the nominal type of E_2 be $\mathtt{S}_2$. By the inductive hypothesis, for every possible result q of E_1 in the environment η_1 there is some possible result r from E_1 in the environment η_2 such that $q \, \mathcal{R}_{\mathtt{S}_1} \, r$; similarly for E_2. By the coercion properties of simulation relations, it $\mathcal{R}_{\mathtt{S}_1} \subseteq \mathcal{R}_{\mathtt{T}}$, so $q \, \mathcal{R}_{\mathtt{T}} \, r$; similarly for E_2. Hence the result follows.

- Suppose the expression is $E_1 \bigtriangledown E_2$, which has nominal type $\mathtt{T}$. The possible results of this expression are the union of those from E_1 and E_2, except that $\perp$ appears only if it is a possible result of both. This is the same as the previous case, except that one must be careful about $\perp$. Suppose, for the sake of contradiction, that there was some q in $\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](A, \eta_1)$ such that q is not related by $\mathcal{R}_{\mathtt{T}}$ to some element of $\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](B, \eta_2)$. By the previous case, if $\mathcal{M}[\![E_1 \sqcap E_2]\!](B, \eta_2)$ is the same as $\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](B, \eta_2)$, then this would be a contradiction; so assume that

  $$\perp \quad \notin \quad \mathcal{M}[\![E_1 \bigtriangledown E_2]\!](B, \eta_2) \tag{7.8}$$

  $$q \quad \mathcal{R}_{\mathtt{T}} \quad \perp. \tag{7.9}$$

  Since $\mathcal{R}_{\mathtt{T}}$ is bistrict, $q = \perp$. Furthermore, by definition of $\bigtriangledown$, it must be that either E_1 or E_2 is guaranteed to terminate in B and η_2. Without loss of generality, suppose

  $$\perp \notin \mathcal{M}[\![E_1]\!](B, \eta_2). \tag{7.10}$$

So by the inductive hypothesis

$$\mathcal{M}[\![E_1]\!](A,\eta_1)\,\mathcal{R}_{\mathtt{T}}\,\mathcal{M}[\![E_1]\!](B,\eta_2). \tag{7.11}$$

Thus $\perp \notin \mathcal{M}[\![E_1]\!](A,\eta_1)$, since $\mathcal{R}$ is bistrict. But then by definition of NOAL's angelic choice operator,

$$\perp \notin \mathcal{M}[\![E_1 \bigtriangledown E_2]\!](A,\eta_1). \tag{7.12}$$

Since $q = \perp$, the above contradicts the assumption that $q \in \mathcal{M}[\![E_1 \bigtriangledown E_2]\!](A,\eta_1)$. Hence the result follows.

- If the expression is `isDef?(`E_1`)`, then the result follows directly from the inductive hypothesis applied to E_1 and the bistrictness of $\mathcal{R}$.

∎

The proof of the above lemma is the source of the requirement that simulation relationships at a subtype also hold at each supertype. This property guarantees that expressions related at a subtype are also related when a function call or a combination exploits subtype polymorphism. For example, if E has nominal type `S`, `S` is a subtype of `T`, and the function identifier f has nominal signature $\mathtt{T} \to \mathtt{U}$, then the expression `f(`E`)` is type-safe; furthermore, if the meanings of E in η_1 and η_2 are related at type `S` and if the meanings of f are also related at $\mathtt{T} \to \mathtt{U}$, then by this coercion property the arguments are related at the nominal argument type `T`, and so the results will be related at the nominal result type `U`.

To show that the substitution property holds for NOAL programs one needs to show that simulation is preserved by recursively-defined NOAL functions. The proof is involved because of NOAL's erratic and angelic choice expressions and has therefore been relegated to Appendix C. The idea of the proof is as follows. To deal with functions that use only erratic choice one uses a family of approximations, each of which is deterministic and that together cover the choices available to functions that use erratic choice. To deal with angelic choice one first rewrites the functions, replacing angelic with erratic choices and uses the limit of the erratic choice approximations as a first approximation. Then one expands recursive calls in-line, obtaining a series of approximations that use angelic choice for deeper and deeper recursions. At each stage of approximation, simulation is preserved. Simulation is also preserved by the various limit operators. This series of lemmas culminates in the following result.

Lemma 7.2.2. Let

$$\begin{aligned}&\mathtt{fun}\ f_1(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_1;\\ &\vdots\\ &\mathtt{fun}\ f_m(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_m\end{aligned}$$

be a mutually recursive system of NOAL function definitions.

Suppose $\mathcal{R}$ is a Σ-simulation relation between Σ-algebras A and B Then for each j from 1 to m,

$$\mathcal{F}[\![f_j]\!](A)\,\mathcal{R}_{\vec{\mathtt{S}_j}\to\mathtt{T}_j}\,\mathcal{F}[\![f_j]\!](B). \tag{7.13}$$

Proof: See Appendix C. ∎

The main result of this section is the following theorem, which says that simulation is a valid criterion for imitation. The significance of the theorem is that a simulation will not allow surprising behavior.

Theorem 7.2.3. Let Σ be a signature. Let $\leq$ be the presumed subtype relation of Σ. Let A and B be Σ-algebras.

If $\mathcal{R}$ is a Σ-simulation relation between A and B, then for all sets of typed identifiers X, for all environments $\eta_A : X \to A$, and for all environments $\eta_B : X \to B$, if $\eta_A\,\mathcal{R}\,\eta_B$, then (A,η_A) imitates (B,η_B) with respect to the set of all type-safe NOAL programs.

Proof: Suppose that $\mathcal{R}$ is a Σ-simulation relation between A and B. Let $X = \{\vec{\mathtt{x}} : \vec{\mathtt{U}}\}$ be a set of typed identifiers and let $\eta_1 : X \to A$ and $\eta_2 : X \to B$ be such that $\eta_1\,\mathcal{R}\,\eta_2$. Let P be a type-safe NOAL program of the form:

$$\begin{aligned}&\mathtt{fun}\ f_1\ (\vec{\mathtt{z}_1} : \vec{\mathtt{S}_1})\mathtt{:}\ \mathtt{T}_1\ \mathtt{=}\ E_1\,\mathtt{;}\\ &\vdots\\ &\mathtt{fun}\ f_m\ (\vec{\mathtt{z}_m} : \vec{\mathtt{S}_m})\mathtt{:}\ \mathtt{T}_m\ \mathtt{=}\ E_m\,\mathtt{;}\\ &\mathtt{program}\ (\vec{\mathtt{x}} : \vec{\mathtt{U}})\mathtt{:T}\ \mathtt{=}\ E_m\,\mathtt{.}\end{aligned}$$

Let Z be the set of typed function identifiers that contains the f_j with their nominal signatures. Let $\eta_1' : Z \cup X \to A$ and $\eta_2' : Z \cup X \to B$ be defined so that for all $\mathtt{x}_i \in X$, $\eta_1'(\mathtt{x}_i) = \eta_1(\mathtt{x}_i)$, $\eta_2'(\mathtt{x}_i) = \eta_2(\mathtt{x}_i)$ and for all $f_j \in Z$, $\eta_1'(f_j)$ is $\mathcal{F}[\![f_j]\!](A)$ and $\eta_2'(f_j)$ is $\mathcal{F}[\![f_j]\!](B)$.

By Lemma 7.2.2, $\eta_1'\,\mathcal{R}\,\eta_2'$, since the denotations of recursively defined functions are related by $\mathcal{R}$. So by Lemma 7.2.1,

$$\mathcal{M}[\![E]\!](A,\eta_1')\,\mathcal{R}_{\mathtt{T}}\,\mathcal{M}[\![E]\!](B,\eta_2'). \tag{7.14}$$

Recall that this means that for each $q \in \mathcal{M}[\![E]\!](A,\eta_1')$, there is some $r \in \mathcal{M}[\![E]\!](B,\eta_2')$ such that $q\,\mathcal{R}_{\mathtt{T}}\,r$. Since P is a program, the nominal type of E must be a visible type; that is, $\mathtt{T} \in V$. By Lemma 5.3.4, each such q and r has type `T`. Since *Visible* is the identity on the visible types,

$$\mathcal{M}[\![E]\!](A,\eta_1') = \mathcal{M}[\![P]\!](A,\eta_1) \tag{7.15}$$

$$\mathcal{M}[\![E]\!](B,\eta_2') = \mathcal{M}[\![P]\!](B,\eta_2). \tag{7.16}$$

Since $\mathcal{R}$ is V-identical, for each $q \in \mathcal{M}[\![P]\!](A,\eta_1)$, there is some $r \in \mathcal{M}[\![P]\!](B,\eta_2)$ such that $q = r$; that is,

$$\mathcal{M}[\![P]\!](A,\eta_1) \subseteq \mathcal{M}[\![P]\!](B,\eta_2). \tag{7.17}$$

Therefore (A,η_1) imitates (B,η_2) with respect to the set of type-safe NOAL programs. ∎

7.3 A Weaker Definition of Subtyping based on Imitation

As in [Lea89], it is possible to give a weaker definition of subtype relations based on imitation. The advantage of the following notion of "weak subtyping" is that it is dependent on a set of observations, and can

thus be tailored more exactly to a given programming language. Furthermore, weak subtyping does not depend on the way that types are specified; that is, the behavior of the specification functions does not have to be preserved by a weak subtype.

Weak subtype relations are based strictly on observable behavior, unlike subtype relations. The idea is that, an instance of a subtype cannot be observed to act differently than an instance of the supertype. Technically this condition is expressed by the requirement that each environment that allows subtyping must imitate some nominal environment, which prevents surprises from the interaction of several objects.

Definition 7.3.1 (weak subtype relation). Let Σ be a signature. Let $SPEC$ be a nonempty collection of Σ-algebras with the same $SIG(B)$-reduct, where B is a fixed algebra that defines the visible types. Let $\leq$ be the presumed subtype relation of Σ. Let OBS be a set of Σ-observations. Then $\leq$ is a *weak subtype relation on the types of $SPEC$ with respect to OBS* if and only if for all algebras $C \in SPEC$, there is some $A \in SPEC$ such that for all sets of typed identifiers X and for all Σ-environments $\eta_C : X \to |C|$, there is some nominal environment $\eta_A : X \to |A|$ such that (C, η_C) imitates (A, η_A) with respect to OBS.

A trivial example of a weak subtype relation is the identity relation on types; the identity relation is always a subtype relation because the imitates relation is reflexive.

Example 7.3.2. Consider the specification II. The presumed subtype relation on II is the smallest reflexive relation on the types of II such that `Interval` $\leq$ `IntSet`. This relation $\leq$ is a weak subtype relation with respect to the following set of observations:

$$\{\mathcal{M}[\![\texttt{program (x:IntSet):Int = size(x)}]\!]\}.$$

To see this, let C be an II-algebra, and let $\eta_C : \{\texttt{x} : \texttt{IntSet}\} \to C$ be a $SIG(II)$-environment. Let A be an II-algebra. Let $\eta_A : \{\texttt{x} : \texttt{IntSet}\} \to A$ be defined such that if $\eta_C(\texttt{x})$ is a proper instance of `Interval`, then $\eta_A(\texttt{x})$ is an instance of `IntSet` or `Interval` such that

$$\text{size}^A(\eta_A(\texttt{x})) = \text{size}^C(\eta_C(\texttt{x})). \tag{7.18}$$

Otherwise, if $\eta_C(\texttt{x}) = \perp$, then let $\eta_A(\texttt{x}) = \perp$. Then (C, η_C) imitates (C, η_2) with respect to the above set of NOAL programs, as is easily checked.

Example 7.3.3. As a counter-example, consider a set of type specifications including `IntSet` and `Interval` such that `IntSet` $\leq$ `Interval`. Call this specification II'. The relation $\leq$ is not a weak subtype relation with respect to the set of type-safe NOAL programs over the $SIG(II')$. To see this, consider the observation that is the denotation of the following NOAL program.

```
program (x:Interval):IntStream =
        choose(x) &
        (choose(x) & empty(IntStream))
```

When this program is applied to an II'-algebra and a nominal environment that maps `x` to a proper value of type `Interval`, the only possible results are streams consisting of two identical integers (e.g., $\langle 2,2 \rangle$ is the only possible result when the result of `create(Interval,2,3)` is bound to `x`). However, if this program is applied to an II'-algebra and an environment where the `IntSet` with abstract value {2,3} is bound to `x`, then the stream $\langle 2,3 \rangle$ is a possible result. Therefore, such an algebra-environment pair does not imitate a nominal algebra-environment pair, and hence this $\leq$ is not a weak subtype relation.

As with the standard definition of subtype relations, weak subtypes can be more deterministic and incompletely specified supertypes can be handled [Lea89, Section 5.1].

Like the imitates relation, whether a binary relation on types is a weak subtype relation varies with the observations one makes. As a trivial example, every binary relation on types is a weak subtype relation with respect to the empty set of observations. One can also show that a binary relation $\leq$ such that `IntSet` $\leq$ `Interval` is a weak subtype relationship with respect to the set of observations

$$\{\mathcal{M}[\![\texttt{program (x:Interval):Int = elem(x,2)}]\!]\}.$$

If a feature is added to one's programming language, then some binary relations on types may cease to be weak subtype relationships with respect to programs written in the enlarged language. However, if one removes a feature from a language, existing weak subtype relations remain valid.

Lemma 7.3.4. Let Σ be a signature. Let $SPEC$ be a set of Σ-algebras. Let OBS and OBS' be sets of Σ-observations.

If $OBS \supseteq OBS'$, then all weak subtype relations on the types of $SPEC$ with respect to OBS are also weak subtype relations with respect to OBS'. ∎

One way to handle an enlarged programming language is suggested by the following lemma. If one knows that $\leq$ is a weak subtype relation on the types of $SPEC$ with respect to OBS_1, then to verify that $\leq$ is a weak subtype relation with respect to $OBS_1 \cup OBS_2$ one merely has to verify that $\leq$ is a weak subtype relation with respect to OBS_2.

Lemma 7.3.5. Let Σ be a signature. Let $SPEC$ be a set of Σ-algebras. Let $OBS = \bigcup_{i \in I} OBS_i$ be a set of Σ-observations.

If for each $i \in I$, $\leq$ is a weak subtype relation on the types of $SPEC$ with respect to OBS_i, then $\leq$ is a weak subtype relation on the types of $SPEC$ with respect to OBS. ∎

Some other thoughts about how certain features of programming languages affect subtype relations are found in Chapter 9.

7.4 Subtype Relations are Weak Subtype Relations for NOAL

The main result of this section is a theorem that states that each subtype relation is a weak subtype relation

with respect to type-safe NOAL programs. This theorem guarantees that a subtype cannot exhibit surprising results in a NOAL program. Another consequence of this theorem is that to prove that a relation is *not* a subtype relation, one need only give a single program that shows that the relation is not a weak subtype relation.

Theorem 7.4.1. Let Σ be a signature. Let $SPEC$ be a nonempty collection of Σ-algebras with the same $SIG(B)$-reduct, where B is a fixed algebra that defines the visible types. Let $\leq$ be the presumed subtype relation of Σ.

If $\leq$ is a subtype relation on the types of $SPEC$, then $\leq$ is a weak subtype relation on the types of $SPEC$ with respect to the set of type-safe NOAL programs over Σ.

Proof: Suppose that $\leq$ is a subtype relation. Let $C \in SPEC$ be given. By definition of subtype relations, there is some $A \in SPEC$ and some Σ-simulation relation $\mathcal{R}$ between C and A. Let X be a set of typed identifiers. Let $\eta_C : X \rightarrow |C|$ be a Σ-environment. Let $\eta_A : X \rightarrow |A|$ be a nominal environment such that $\eta_C \, \mathcal{R} \, \eta_A$. The environment η_A must exist, because of the coercion property of a simulation relation. That is, for each $\mathtt{x} : \mathtt{T} \in X$, there is some type $\mathtt{S} \leq \mathtt{T}$ such that $\eta_C(\mathtt{x}) \in \mathtt{S}^C$. By the coercion property, there is some $r \in \mathtt{T}^A$ such that $\eta_C(\mathtt{x}) \, \mathcal{R} \, r$. So let $\eta_A(\mathtt{x})$ be r.

Since $\eta_C \, \mathcal{R} \, \eta_A$, by Theorem 7.2.3, (C, η_C) imitates (A, η_A) with respect to the set of type-safe NOAL programs. So by definition, $\leq$ is a weak subtype relation. ∎

7.5 Discussion

7.5.1 Testing

Since each subtype relation is a weak subtype relation with respect to NOAL programs, it follows that one can disprove a subtype relation by showing that it is not a weak subtype relation. That is, a test that shows that a binary relation on types is not a weak subtype relation automatically shows that it is not a subtype relation. This is often easier than a direct proof.

If a test of a program that uses subtype polymorphism reveals an error (a surprising result), then there can be several problems:

- the program logic is incorrect,
- the presumed subtype relation is not a subtype relation, or
- the implementation of some type is incorrect.

However, it is not necessary to test the program's logic using subtypes of the types explicitly mentioned in the program. This is because the supertypes can exhibit all the possible behaviors, subtypes can only exhibit a subset of the behaviors of their supertypes. So if an error can be revealed by testing, then there is some implementation of the types explicitly used in the program that can reveal it.

7.5.2 Comparing the two Notions of Subtyping

The notions of imitation and weak subtyping play a crucial role in my dissertation [Lea89], where weak subtype relations are called subtype relations. The reasons for favoring an algebraic definition of subtype relations based on simulation over a definition based on observations and imitation in the present work are the following.

- Only subtyping based on simulation is strong enough to prove the soundness of the modular verification techniques that use nominal type information described in Chapter 6. This was true even in my dissertation.
- The semantics of specifications are vastly simplified by requiring that subtypes also interpret the specification functions of their supertypes. In contrast, the semantics of specifications in my dissertation are highly non-standard and considerable effort is spent to show that the semantics is well-defined. Even so, assertions in the specifications of my dissertation must be program-observable as opposed to merely subtype-constraining, as discussed in Chapter 3. By extending the notion of simulation described in my dissertation to encompass specification functions as well as program operations, the proof of the soundness of the verification system becomes much simpler.
- Since the algebraic relationships of simulation and subtype relations are not context dependent, they are simpler, which has simplified the presentation of this report compared with my dissertation.

On the other hand, weak subtype relations may be better for informal reasoning, since they are not dependent on the way that types are specified. As in my dissertation, such informal specifications should be confined to using program-observable assertions.

Chapter 8

Discussion

The discussion in this chapter covers what extensions to the formal apparatus are needed for practical applications, and future work.

8.1 Extensions Needed for Practical Applications

At least two extensions of the results in this report are needed if they are to be directly used in "real" languages such as Smalltalk-80 or Trellis/Owl. These extensions would extend the definition of subtype relations to other kinds of types and would give designers a technique for proving subtype relations.

The most important extension would be to adapt the results to languages with mutable types and aliasing. The algebraic models presented above are only suited for modeling immutable types, but most object-oriented programs make heavy use of mutation. Dealing with mutation and aliasing will also complicate the logic used for verification.

Parameterized types, such as `Set[T]` are also not considered above. This is not a severe limitation, however, since one can describe subtype relations and reasoning for instantiations of parameterized types. Still, it would be interesting to describe the subtype relationships among parameterized types more directly and to use such relationships to derive subtype relationships on their instantiations[1].

8.2 Future Work

This section describes future work in the areas of specification, verification, and language design.

8.2.1 Future Work on Specification

Inheritance of specifications by subtypes is attractive and would help make a practical specification language.

Another question is whether one can factor proofs of subtype relationships by taking advantage of controlled inheritance of specifications. For example, if the specification of a type `S` incorporates the specification of a type `T`, then it should be possible to take advantage of this relationship when proving that `S` is a subtype of `T`.

There is also work to be done in overcoming the limitations of the specification language in describing the effect of functions on subtypes. These limitations are discussed in Chapter 3.

Finally, it would be helpful to be able to derive traits for subtypes in a more automatic fashion.

8.2.2 Future Work on Verification

An important extension to the verification techniques would be to support the verification of modules that implement abstract types (classes). There are two aspects to this problem. The first involves showing that a class meets the specification of the type it purports to implement in the presence of subtype polymorphism. The second aspect is how to factor the proof of correctness for a subclass to take advantage of the proof of correctness of its superclasses.

The other side of the verification problem is that of verifying that a specified relation on types is a subtype relation. The idea of specifying the simulation relation along with a set of type specifications should help, but a symbolic technique for this verification is needed. The direct use of the definition of subtype relations, by constructing the algebras and simulation relations, is too mathematically taxing to be generally useful. One approach to such a result is to work with implications between the pre- and post-conditions of operation specifications (and invariants), following the lead of Meyer [Mey88] or America [Ame89]. It might also be useful to have a proof-theoretic definition of subtype relations. That is, how can one characterize subtype relations using the set of valid assertions that can be made about the objects of various abstract types?

Another set of important questions for the verification of subtype relations concerns how to prove subtype relationships in a modular fashion. If one adds a new type specification to an existing design, the following questions arise.

- What conditions on the specification of the new type will ensure that the old subtype relation is still a subtype relation?
- What has to be shown to add new subtype relationships involving the new type to the old subtype relation?

8.2.3 Future Work on Language Design

One long-range project would be to design an object-oriented programming language that would support subtype polymorphism, subtyping, inheritance, and program verification. Such a language should have a type system that can ensure obedience to a subtype

[1] A related question is what kind of parameterization is necessary or useful in a language with subtype polymorphism.

relation. However, it is too early to tell what other features a language would need to support program verification. For example, it is not yet known what features of an inheritance mechanism help or hinder verification.

A more modest language design project would be to solve the name-clash (or interface control) problem for languages with message passing mechanisms [LL85]. In a language with message passing, each object's instance operations form a behavioral interface that is analogous to the behavioral interface of an abstract type. However, in all languages with message passing mechanisms, there is no way to change an object's interface. Therefore each object presents the same interface to all parts of a program (except for the class that implements the object's behavior). It can be difficult and costly to combine independently designed program parts that assume that the same instance operation name means different things.

Furthermore, subtyping depends on object interfaces. For example, a type `Interval2` whose objects behave like `Interval`s, but has an instance operations named `pick` instead of `choose` will not be a subtype of `IntSet` even though instances of type `Interval2` otherwise behave like instances of `IntSet`.

A mechanism to mediate between independently designed abstractions with fixed interfaces is a feature of several languages without message passing mechanisms (e.g., OBJ2 [FGJM85], Argus [LDH+87], and Ada [Ada83]), where one can change the interface of a type parameter. In a language with a message passing mechanism, one wants to be able to change the interfaces of *objects*. The ability to change object interfaces could also be exploited to provide access control for objects [JL76] [JL78].

Chapter 9

Summary and Conclusions

A high-level summary of results and their significance is offered in this chapter, as well as some conclusions about programming and programming language design.

9.1 Summary of Results

The two main results in this report are a new definition of subtype relations and new techniques for the modular specification and verification of object-oriented programs that use subtype polymorphism.

The precise definition of subtype relations embodies the intuition that each instance of a subtype simulates some instance of that type's supertypes. So programs can manipulate instances of a subtype as if they were instances of that type's supertypes without surprising results.

The most important property of the definition of subtype relations is that it allows *abstract* types to be compared, based on their specifications. Technically, this is because the definition of subtype relations is based on the semantics of specifications. Most other work on subtyping only describes subtype relationships for a fixed set of built-in types (e.g., [Car84]). The definition of subtype relations also allows incompletely specified and nondeterministic types to be compared, so it is more widely applicable than Bruce and Wegner's definition [BW87a].

Simulation as defined in this report handles nondeterminism as follows. A nondeterministic object q simulates an object r if q has only the behaviors that r has; however, q may be more deterministic.

The definition of subtype relations takes the potential incompleteness of specifications into account as follows. For any given implementation of the subtype, there must be some implementation of the supertype such that each object of the subtype simulates some object of the supertype, for those implementations. This definition applies even to specifications for which no single implementation captures all the permitted behavior.

Modular specification is ensured by requiring that the specification functions used to specify a supertype also apply to subtypes. The meanings of such specification functions must be preserved by subtypes.

Finally, modular verification of NOAL programs is possible because of the semantic restrictions on subtype relations. The verification is modular in that one can verify programs using static type information, without explicit concern for possible subtypes. Moreover, one can add new types to a program without updating the verification. The only significant difference from standard program verification is that the verifier must also show that the specified subtype relation has the necessary semantic properties.

9.2 Conclusions for Programmers

The significance of the results in this report and some lessons for programmers who work with object-oriented programming languages that have message passing are described in this section.

Subtype relations are a new tool for programmers. Subtype relationships are similar to refinement relationships among abstract types; the difference is that the syntax of class operations does not matter for a subtype relationship. Subtype relations are useful during program design, where they can help track the evolution of abstractions, limit the effects of specification changes, and group and classify related types [Lis88]. In a system like Smalltalk-80 where classes are also objects, subtype relationships among the types of classes (metatypes) can also be used in similar ways. Subtype relations can be used to write polymorphic specifications and to support careful reasoning.

Perhaps the most important lesson for programmers is the most basic one: subtype relationships are based on specified behavior and they have nothing to do with how a type is implemented [Sny86a]. That is, a subtype is *not* a subclass. While it is useful to record inheritance relationships among implementations in a subclass relation, one should organize abstract types by a subtype relation. This distinction between subclasses and subtypes, when properly understood, can be a powerful tool for separation of concerns. Subtype relations allow one to reason abstractly about instances of abstract types.

The distinction between subtypes and subclasses is not just academic. If one passes an argument whose type is not a subtype of the expected formal argument type to a procedure, one has no guarantee that the procedure will act as desired. If one uses an instance of a subclass where instances of a superclass are expected, then one's programs may behave in unexpected ways. To prevent such problems one should ensure that each expression denotes an object whose type is a subtype of the expression's nominal type. If one programs in a statically type-checked language like Trellis/Owl, then the type system can check this second property automatically, once it has been told about a subtype relation.

9.3 Conclusions for Language Designers

Some lessons for designers of new programming languages with message passing mechanisms are discussed in this section.

9.3.1 Languages Should Have Declared Subtype Relations

If one is designing a type system for an object-oriented programming language with a message passing mechanism, then subtype relations should be a part of that type system. (Otherwise programs will not be able to exploit subtype polymorphism.) Perhaps the most important lesson for language designers is to make the programmer declare the subtype relation for abstract types.

The reason this lesson is so important is that the programming language cannot, in general, find a nontrivial subtype relation on the types of a program. Most programming languages are not designed to include behavioral specifications as part of programs. Each module is a specification of that module's behavior, but it is not the specification that the programmer worked from during design (and verification). Even if the program text included a behavioral specification, the problem of finding a nontrivial subtype relation on the types of a specification is undecidable in general. It seems more straight-forward to let the programmer declare a subtype relation. Finally, a programmer may wish to work in a subset of a full language, and thus may only be concerned with weak subtype relations with respect to that subset of programs.

9.3.2 TypeOf Operators Cause Problems for Reasoning

Another lesson for language designers is that operators that tell the type of an object cause problems for reasoning and should be avoided. This is a new twist on an old lesson: if one wants to reason about abstract types based on their specifications, then one's language should only allow objects to be observed by invoking their instance operations.

A `typeOf` operator returns the type of an object as a string. For example, the program `typeOf(x)` will give different results in environments where `x` denotes objects of different types. Such an operator destroys subtyping. It is easy to show that a weak subtype relation with respect to the set of all programs that use `typeOf` cannot relate different types. So other methods must be used to reason about parts of an implementation that use a `typeOf` operator.

Appendix A

Summary of Notation

Notation used in earlier chapters is summarized in this appendix. The definitions given in earlier chapters are summarized by boolean functions (what is a "subtype relation"?), whose signatures are given here.

Table A.1 lists some primitive domains, which are just sets. Algebras are heterogeneous (i.e., sorted), so one should think of Object as the disjoint union of several sets (one for each sort). In this appendix the carrier sets of various sorts are not distinguished for the sake of simplicity. The notation $\vec{q}$ means a tuple of objects, possibly empty.

As in Table A.1, phrases are often abbreviated. For example, "ProgOpSym" should be read as "program operation symbol."

The syntax of the terms in the specification language is given in Figure 3.1. The syntax of NOAL expressions is given in Figure 5.1.

The following tables are organized by topic, which is roughly by chapter, except that the syntax and semantics of specifications and programs are treated separately. For each topic there are one or two tables. One table is organized like Table A.1 and describes the domains related to that topic. The second table lists the significant questions (i.e., definitions) related to that topic.

The following conventions are used to describe domains. Each entry has the form $d \in D = E'$ meaning that d is the typical notation for an element of the domain D, which is defined by E'. For example

$$\eta \in \mathrm{Env} = \mathrm{TypedIdent} \rightarrow \mathrm{Object}$$

means that η is used to denote environments, which are mappings from typed identifiers to objects. The following notations are used in describing domains. The notation $\{D\}$ means a nonempty set of elements from the domain named D. The notation D^* stands for all finite tuples of zero or more Ds. The notation $D_1 \rightarrow D_2$ denotes the set of functions from a subset of D_1 to D_2; that is, partial functions from D_1 to D_2. The notation (D_1, D_2) stands for the set of all pairs whose first element is from D_1 and whose second element is from D_2. The notation $\{(D_1, D_2)\}$ stands for a binary relation between D_1 and D_2.

Table A.2 describes signatures, algebras and some related operations from Chapter 2. The notation $A_{(\Sigma)}$, where Σ is a signature, means the Σ-reduct of the algebra A. The notation $\mathtt{T}^A$ means the carrier set of $\mathtt{T}$ in the algebra A. If q is an object, the notation $q : \mathtt{T}$ means $q \in \mathtt{T}^A$. The notation $\vec{\mathtt{S}}^A$ means a tuple of carrier sets, and $\vec{q} \in \vec{\mathtt{S}}^A$ means that $q_i \in \mathtt{S}_i{}^A$, for each i. If Q and R are sets, the notation $Q \,\mathcal{R}_{\mathtt{T}}\, R$ means that for each $q \in Q$ there is some $r \in R$ such that $q \,\mathcal{R}_{\mathtt{T}}\, r$. The notation $\vec{q}\,\mathcal{R}_{\vec{\mathtt{S}}}\,\vec{r}$ means that for each i, $q_i \,\mathcal{R}_{\mathtt{S}_i}\, r_i$.

Table A.3 describes questions for algebras and the definition of simulation relations.

Table A.4 describes the concepts used to describe the syntax and semantics of the specification language of Chapter 3. The structure of Larch traits is not further described. The notation $\overline{\eta}$ denotes the extension of the environment η to a mapping from terms to the elements of the carrier set of the algebra in the range of η. If the environments η and η' have the same domain, then the notation $\eta \,\mathcal{R}\, \eta'$ means that for each identifier $\mathtt{x} : \mathtt{T}$ in the domain, $\eta(\mathtt{x}) \,\mathcal{R}_{\mathtt{T}}\, \eta'(\mathtt{x})$.

Table A.5 describes the definitions of satisfaction and related concepts from in Chapter 3.

Table A.6 describes the concepts used to define observations in Chapter 7.

Table A.7 describes the concepts used in Chapter 5 and Appendix C to give semantics to NOAL programs and to describe the NOAL type system. The operator that takes the closure of a set is written as an overbar; that is, the closure of a set Q is written $\overline{Q}$.

Table A.8 describes the major definitions of Chapter 5 and Appendix C.

Table A.9 describes subtype relations from Chapter 4 and related concepts from Chapter 7.

Table A.10 describes the concept of a Hoare-triple from Chapter 6 and table A.11 describes related definitions.

Notation for Members	Name	description
o, q, r $\in$	Object	instances
$\mathtt{S}$, $\mathtt{T}$ $\in$	Sort	sort symbols
$\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ $\in$	Identifier	identifiers
$f \in$	FunIdent	NOAL function identifiers
$\mathtt{g} \in$	ProgOpSym	program operation symbols
$\mathrm{f} \in$	SpecFunSym	specification function symbols
true, *false* $\in$	Bool	the booleans
P, Q, R, $\in$	Term	logical formulas
$E \in$	Expr	programming language expressions

Table A.1: Primitive Domains

$SORTS \in$ SetOfSorts	$=$	{Sort}
$TYPES \in$ SetOfTypes	$=$	{Sort}
$V \in$ VisibleTypes	$=$	{Sort}
$\leq \in$ PreordSort	$=$	{(Sort, Sort)}
$SFUNS \in$ SetOfSpecFunSym	$=$	{SpecFunSym}
$POPS \in$ SetOfProgOpSym	$=$	{ProgOpSym}
$OPS \in$ OpSyms	$=$	{SpecFunSym} $\cup$ {ProgOpSym}
$ResSort \in$ ResultSortMap	$=$	OPS, Sort$^* \rightarrow$ Sort
$\Sigma \in$ Signature	$=$	$\left(\begin{array}{l}\text{SetOfSorts, SetOfTypes, VisibleTypes,}\\ \text{PreordSort, SetOfSpecFunSym,}\\ \text{SetOfOpSym, ResultSortMap}\end{array}\right)$
$o, q, r \in$ CarrierSet	$=$	Object
$Q, R \in$ SetOfPossRes	$=$	{Object}
$\mathtt{g}^A \in$ Operation	$=$	Object$^* \rightarrow$ SetOfPossRes
$\mathrm{f}^A \in$ SpecFun	$=$	Object$^* \rightarrow$ Object
$A, B, C \in$ Algebra	$=$	(CarrierSet, {SpecFun}, {Operation})
$A \in$ TraitStruct	$=$	(CarrierSet, {SpecFun})
$SPEC \in$ SpecSemantics	$=$	{Algebra}
$\mathcal{R} \in$ FamilyOfRel	$=$	{{(Object, Object)}}

Table A.2: Algebras and Related Concepts

has sort?	: Object, Algebra, Sort $\rightarrow$ Bool
simulation rel?	: Signature, Algebra, Algebra, FamilyOfRel $\rightarrow$ Bool

Table A.3: Questions for Algebras

P, Q, R	$\in$ Assert	= Term
$\vec{\mathtt{S}} \rightarrow \mathtt{T}$	$\in$ NomSig	= (Sort*, Sort)
$\mathtt{g}$	$\in$ OpSpec	= (ProgOpSym,NomSig,Assert,Assert)
$SPEC$	$\in$ SetOfTypeSpec	= (SetOfTypes, PreordSort, {Trait}, {OpSpec})
SIG	$\in$ SigOfSpec	= SetOfTypeSpec $\rightarrow$ Signature
$\mathtt{x} : \mathtt{T}$	$\in$ TypedIdent	= (Identifier, Sort)
η	$\in$ Env	= TypedIdent $\rightarrow$ Object
$\overline{\eta}$	$\in$ ExtendedEnv	= Term $\rightarrow$ Object
	$\in$ FunSpec	= (FunIdent,NomSig,SetOfTypeSpec,Assert,Assert)
$FSPEC$	$\in$ SetOfFunSpec	= {FunSpec}
SIG	$\in$ SigOfFunSpec	= SetOfFunSpec $\rightarrow$ (FunIdent $\rightarrow$ NomSig)
f	$\in$ FunImpl	= Algebra $\rightarrow$ (Object* $\rightarrow$ SetOfPossRes)

Table A.4: Specifications and Related Concepts

nominal?	: Algebra, Env $\rightarrow$ Bool
proper?	: Env $\rightarrow$ Bool
models?	: Algebra, Env, Assert $\rightarrow$ Bool
satisfies?	: Algebra, SetOfTypeSpec $\rightarrow$ Bool
satisfies?	: FunImpl, FunSpec, SetOfTypeSpec $\rightarrow$ Bool

Table A.5: Questions for Specifications

P	$\in$ Observation	= Algebra, Env $\rightarrow$ SetOfPossRes
OBS	$\in$ SetOfObs	= {Observation}

Table A.6: Observations and Related Concepts

X, Y, Z	$\in$ SetOfIdent	= {TypedIdent}
$\mathcal{M}$	$\in$ Denotation	= Expr $\rightarrow$ Observation
$\mathcal{F}$	$\in$ FunDenotation	= FunDef $\rightarrow$ FunImpl
H, X	$\in$ TypeAssumptions	= {TypedIdent}
$\sqsubseteq$	$\in$ DomainOrder	= {(Object, Object)}
$\sqsubseteq_E$	$\in$ DomOrdForSets	= {({Object}, {Object})}

Table A.7: Programming Language Concepts

nominal type?	: Expr, Signature $\rightarrow$ Sort
monotonic?	: Operation $\rightarrow$ Bool
strongly monotonic?	: Operation $\rightarrow$ Bool
continuous?	: Operation $\rightarrow$ Bool

Table A.8: Questions for Programming Language

subtype relation?	: SpecSemantics, PreordSort → Bool
weak subtype relation?	: SpecSemantics, PreordSort, SetOfObs → Bool
imitates?	: Algebra, Env, Algebra, Env, SetOfObs → Bool

Table A.9: Subtype Relations and Related Concepts

R, P	∈ PreCond	= Assert
Q	∈ PostCond	= Assert
$P\ \{\mathtt{y} \leftarrow E\}$	Q ∈ HoareTriple	= (PreCond,Identifier,Expression,PostCond)

Table A.10: Verification Concepts

models?	: Algebra, Env, HoareTriple → Bool
valid?	: SpecSemantics, HoareTriple → Bool
provable?	: SetOfTypeSpec, HoareTriple → Bool

Table A.11: Questions for Verification

Appendix B

Visible Types and Streams

The models of the visible types fixed by the type specification language of Chapter 3 are described in this appendix. These types are `Bool`, `Int` and two corresponding stream types: `BoolStream` and `IntStream`.

The model of the type `Bool` is found in Figure B.1.

The model of the type `Int` is found in Figure B.2.

The types `IntStream` and `BoolStream` are used to model output. The `cons` operation of each type is lazy; that is, `cons` is not strict in its second argument. Figure B.3 is an algebraic model `IntStream`. The model of `BoolStream` is similar and can be obtained by replacing `Bool` for `Int` throughout.

The carrier set of `IntStream` is defined using the operator *Stream* [Bro86], defined as

$$Stream(I) \stackrel{\text{def}}{=} \{I^* \cup (I^* \times \{\bot\}) \cup I^\infty\}, \qquad \text{(B.1)}$$

where

- I^* denotes the set of *finite streams*, which are finite sequences of elements of I, such as the empty stream $\langle\rangle$ and $\langle i_1, i_2, i_3\rangle$,
- $I^* \times \{\bot\}$ denotes the set of *partial streams*, which are finite sequences ending in $\bot$, such as $\langle i_1, i_2, i_3, \bot\rangle$ and the totally undefined stream $\bot = \langle\bot\rangle$, and
- I^∞ denotes the set of *infinite streams*, such as $\langle i_1, i_2, i_3, \ldots\rangle$.

The definition of the `rest` operation also needs some explanation. The `rest` operation is strict, as Figure B.3 shows, since all specification functions are strict. Furthermore, one should think of the `rest` operation as requiring that its argument stream not be empty, since the set of possible results of invoking `rest` on an empty stream is the entire carrier set of `IntStream`. Finally, note that the `cons` operation is not strict in its stream argument, as this is how partial streams are constructed.

Carrier Sets

$$\begin{array}{lcl} \texttt{Bool}^B & \stackrel{\text{def}}{=} & \{\bot, true, false\} \\ \texttt{BoolClass}^B & \stackrel{\text{def}}{=} & \{\bot, Bool\} \end{array}$$

Specification Functions

$$\begin{array}{lcl} \text{Bool}^B() & \stackrel{\text{def}}{=} & Bool \\ \text{true}^B() & \stackrel{\text{def}}{=} & true \\ \text{false}^B() & \stackrel{\text{def}}{=} & false \\ \neg\#^B(b) & \stackrel{\text{def}}{=} & \begin{cases} false & \text{if } b = true \\ true & \text{if } b = false \end{cases} \\ \#\&\#^B(b_1, b_2) & \stackrel{\text{def}}{=} & \begin{cases} true & \text{if } b_1 = b_2 = true \\ false & \text{otherwise} \end{cases} \\ \#|\#^B(b_1, b_2) & \stackrel{\text{def}}{=} & \begin{cases} false & \text{if } b_1 = b_2 = false \\ true & \text{otherwise} \end{cases} \\ \# \Rightarrow \#^B(b_1, b_2) & \stackrel{\text{def}}{=} & \begin{cases} false & \text{if } b_1 = true \\ & \text{and } b_2 = false \\ true & \text{otherwise} \end{cases} \\ \# \equiv \#^B(b_1, b_2) & \stackrel{\text{def}}{=} & \begin{cases} true & \text{if } b_1 = b_2 \\ false & \text{otherwise} \end{cases} \end{array}$$

Program Operations

$$\begin{array}{lcl} \texttt{Bool}^B() & \stackrel{\text{def}}{=} & \{Bool\} \\ \texttt{true}^B(Bool) & \stackrel{\text{def}}{=} & \{true\} \\ \texttt{false}^B(Bool) & \stackrel{\text{def}}{=} & \{false\} \\ \texttt{not}^B(b) & \stackrel{\text{def}}{=} & \{\neg\#^B(b)\} \\ \texttt{and}^B(b_1, b_2) & \stackrel{\text{def}}{=} & \{\#\&\#^B(b_1, b_2)\} \\ \texttt{or}^B(b_1, b_2) & \stackrel{\text{def}}{=} & \{\#|\#^B(b_1, b_2)\} \end{array}$$

Figure B.1: Model of the visible type `Bool`.

Carrier sets

$$\begin{array}{lcl} \texttt{Int}^B & \stackrel{\text{def}}{=} & \{\bot, 0, 1, -1, 2, -2, \ldots\} \\ \texttt{IntClass}^B & \stackrel{\text{def}}{=} & \{\bot, \mathit{Int}\} \end{array}$$

Specification Functions

$$\begin{array}{lcl} \mathrm{Int}^B() & \stackrel{\text{def}}{=} & \mathit{Int} \\ 0^B() & \stackrel{\text{def}}{=} & 0 \\ 1^B() & \stackrel{\text{def}}{=} & 1 \\ \#+\#^B(i,j) & \stackrel{\text{def}}{=} & i+j \\ \#-\#^B(i,j) & \stackrel{\text{def}}{=} & i-j \\ -\#^B(i) & \stackrel{\text{def}}{=} & -i \\ \#*\#^B(i,j) & \stackrel{\text{def}}{=} & i \cdot j \\ \#.\mathrm{eq}\#^B(i,j) & \stackrel{\text{def}}{=} & \begin{cases} \mathit{true} & \text{if } i = j \\ \mathit{false} & \text{otherwise} \end{cases} \\ \#<\#^B(i,j) & \stackrel{\text{def}}{=} & \begin{cases} \mathit{true} & \text{if } i < j \\ \mathit{false} & \text{otherwise} \end{cases} \\ \#\leq\#^B(i,j) & \stackrel{\text{def}}{=} & \#|\#^B(\#<\#^B(i,j), \#.\mathrm{eq}\#^B(i,j)) \\ \#>\#^B(i,j) & \stackrel{\text{def}}{=} & \#<\#^B(j,i) \\ \#\geq\#^B(i,j) & \stackrel{\text{def}}{=} & \#\leq\#^B(j,i) \end{array}$$

Program Operations

$$\begin{array}{lcl} \texttt{Int}^B() & \stackrel{\text{def}}{=} & \{\mathit{Int}\} \\ \texttt{one}^B(\mathit{Int}) & \stackrel{\text{def}}{=} & \{1\} \\ \texttt{add}^B(i,j) & \stackrel{\text{def}}{=} & \{\#+\#^B(i,j)\} \\ \texttt{neg}^B(i) & \stackrel{\text{def}}{=} & \{-\#^B(i)\} \\ \texttt{sub}^B(i,j) & \stackrel{\text{def}}{=} & \{\#-\#^B(i,j)\} \\ \texttt{mul}^B(i,j) & \stackrel{\text{def}}{=} & \{\#*\#^B(i,j)\} \\ \texttt{equal?}^B(i,j) & \stackrel{\text{def}}{=} & \{\#.\mathrm{eq}\#^B(i,j)\} \\ \texttt{lt?}^B(i,j) & \stackrel{\text{def}}{=} & \{\#<\#^B(i,j)\} \end{array}$$

Figure B.2: Model of the visible type `Int`.

Carrier sets

$$\begin{array}{lcl} \texttt{IntStream}^B & \stackrel{\text{def}}{=} & \mathit{Stream}(\{0, 1, -1, \ldots, \}) \\ \texttt{IntClass}^B & \stackrel{\text{def}}{=} & \{\bot, \mathit{IntStream}\} \end{array}$$

Specification Functions

$$\begin{array}{lcl} \mathrm{IntStream}^B() & \stackrel{\text{def}}{=} & \mathit{IntStream} \\ \mathrm{empty}^B() & \stackrel{\text{def}}{=} & \langle\rangle \\ \mathrm{cons}^B(s,i) & \stackrel{\text{def}}{=} & \begin{cases} \langle i \rangle & \text{if } s = \langle\rangle \\ \langle i, i_1, \ldots \rangle & \text{if } s = \langle i_1, \ldots \rangle \end{cases} \\ \mathrm{first}^B(s) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } s = \langle\rangle \\ i_1 & \text{if } s = \langle i_1, \ldots \rangle \end{cases} \\ \mathrm{rest}^B(s) & \stackrel{\text{def}}{=} & \begin{cases} \langle\rangle & \text{if } s = \langle\rangle \\ \langle\rangle & \text{if } s = \langle i, \bot \rangle \\ \langle i_1, \ldots \rangle & \text{if } s = \langle i, i_1, \ldots \rangle \end{cases} \\ \mathrm{isEmpty?}^B(s) & \stackrel{\text{def}}{=} & \begin{cases} \mathit{true} & \text{if } s = \langle\rangle \\ \mathit{false} & \text{if } s = \langle i_1, \ldots \rangle \end{cases} \end{array}$$

Program Operations

$$\begin{array}{lcl} \texttt{IntStream}^B() & \stackrel{\text{def}}{=} & \{\mathit{IntStream}\} \\ \texttt{empty}^B(\mathit{IntStream}) & \stackrel{\text{def}}{=} & \{\langle\rangle\} \\ \texttt{undefined}^B(\mathit{IntStream}) & \stackrel{\text{def}}{=} & \{\bot\} \\ \texttt{first}^B(s) & \stackrel{\text{def}}{=} & \begin{cases} \{\mathrm{first}^B(s)\} \\ \quad \text{if } s \neq \langle\rangle \\ \{\bot, 0, 1, -1, \ldots\} \\ \quad \text{if } s = \langle\rangle \end{cases} \\ \texttt{rest}^B(s) & \stackrel{\text{def}}{=} & \begin{cases} \{\mathrm{rest}^B(s)\} \\ \quad \text{if } s \neq \langle\rangle \\ \quad \text{and } s \neq \langle i, \bot \rangle \\ \{\bot\} \\ \quad \text{if } s = \langle i, \bot \rangle \\ \texttt{IntStream}^B \\ \quad \text{if } s = \langle\rangle \end{cases} \\ \texttt{cons}^B(s,i) & \stackrel{\text{def}}{=} & \{\mathrm{cons}^B(s,i)\} \\ \texttt{cons}^B(\bot,i) & \stackrel{\text{def}}{=} & \{\langle i, \bot \rangle\} \\ \texttt{isEmpty?}^B(s) & \stackrel{\text{def}}{=} & \{\mathrm{isEmpty?}^B(s)\} \end{array}$$

Figure B.3: Model of the visible type `IntStream`.

Appendix C

Recursively-Defined NOAL Functions

The semantics of systems of mutually recursive NOAL function definitions are defined in this appendix. This appendix also contains the proof of the substitution property for NOAL functions.

C.1 Semantics of NOAL Functions

The semantics of NOAL functions are discussed informally in Chapter 5. Thus only the formal details are presented here. The semantics follows Broy's discussion of the semantics of AMPL [Bro86, Page 20]. As a preliminary to the semantics, the first subsection below describes how the carrier sets of an algebra are viewed as a domain, which requires an assumption about the domain ordering on the carrier sets of algebras that can be observed by NOAL programs.

In what follows, fix a signature

$$\Sigma = \left(\begin{array}{l} SORTS, TYPES, V, \leq, \\ SFUNS, POPS, ResSort \end{array} \right).$$

C.1.1 Domains and Domain Orderings

The semantics of recursive function definitions require the use of a partial order $\sqsubseteq$ on each type's carrier set that makes that carrier set a pointed complete partial order (i.e., a domain).

The following definition of a pointed complete partial order is taken from [Sch86, Page 111]. For a partially ordered set D, a subset Q of D is a *chain* if it is nonempty and for all $q_1, q_2 \in Q$, either $q_1 \sqsubseteq q_2$ or $q_2 \sqsubseteq q_1$. A *complete partial order* is a set D with a partial order $\sqsubseteq$, such that every chain in D has a least upper bound in D. The *least upper bound* of a chain $Q \subseteq D$, written $\mathrm{lub}(Q)$, is the smallest element of D that is at least as large as every element of Q. A *pointed complete partial order* is a complete partial order that has a least element, $\perp$.

From now on pointed complete partial orders will be called *domains*. Of primary interest are flat domains, since the semantics for recursive functions assumes that each carrier set, except for the carrier sets of the stream types, is a flat domain [Bro86, Page 7].

Definition C.1.1 (flat domain). A domain is *flat* if and only if for all elements q and r, $q \sqsubseteq r$ if and only if $q = r$ or $q = \perp$.

As usual, the notation $q \sqsubset r$ means $q \sqsubseteq r$ and $q \neq r$. Therefore, in a flat domain, $q \sqsubset r$ if and only if $q = \perp$.

The assumptions about the partial order $\sqsubseteq$ on the carrier set of a Σ-algebra A are as follows. Recall that the semantics of the visible types is fixed by convention; that is, the same reduct is used in all algebras for the visible types. It is assumed that `Bool` is a visible type with proper elements *true* and *false*; these are needed to define `if` expressions. It is assumed that for each visible type v, the carrier set of v comes equipped with a partial order $\sqsubseteq$ (defined by convention) that makes v^A a domain with $\perp$ as its least element. It is further assumed that the carrier set of each visible type except `BoolStream` and `IntStream` is a flat domain. For a non-visible type `T`, the ordering $\sqsubseteq$ is defined so that the carrier set of `T` is a flat domain. It is assumed that proper elements of visible types are not related by $\sqsubseteq$ to proper elements of other types and vice versa. (Thus, an algebra may not contain a non-visible type with a carrier set that directly includes proper elements of a visible type.)

The partial ordering $\sqsubseteq$ for the visible types `BoolStream` and `IntStream` is as follows. Let A be an algebra and $q, r \in \texttt{BoolStream}^A$. Then $q \sqsubseteq r$ if and only if either $q = r$ or q is a partial stream whose proper elements are a prefix of r; similarly for `IntStream` [Bro86, Section 2.1].

The carrier set of A itself is a domain formed by the union of all its carrier sets. That is, $q \sqsubseteq r$ in A if and only if q and r are in the same carrier set and $q \sqsubseteq r$. (Recall that there is a single $\perp$ that is in each type's carrier set.)

The partial order $\sqsubseteq$ is extended to tuples as follows: $\vec{q} \sqsubseteq \vec{r}$ if and only if for all i, $q_i \sqsubseteq r_i$.

For sets of possible results, the ordering $\sqsubseteq_E$ is defined so that $Q \sqsubseteq_E R$ if for each $q \in Q$ there is some $r \in R$ such that $q \sqsubseteq r$ [Bro86, Page 13].

For the domain ordering on an algebra to be useful, it must say something about the operations of the algebra. In particular, the operations (both the program operations and the specification functions) of the algebra must be monotonic and continuous.

Definition C.1.2 (monotonic). An operation g is *monotonic* if and only if for all $\vec{q}_1$, $\vec{q}_2$, if $\vec{q}_1 \sqsubseteq \vec{q}_2$, then $g(\vec{q}_1) \sqsubseteq_E g(\vec{q}_2)$.

That is, g is monotonic if whenever $\vec{q}_1 \sqsubseteq \vec{q}_2$ and $r_1 \in g(\vec{q}_1)$, then there is some $r_2 \in g(\vec{q}_2)$ such that $r_1 \sqsubseteq r_2$.

To define continuous operations, chains are viewed as sequences. A *sequence in* $\sqsubseteq$ is a nonempty set $Q = \{q_i \mid i \in I\}$ indexed by some well-ordered set I (whose elements are ordered by $\leq$) with the property that, if $i \leq j$, then $q_i \sqsubseteq q_j$. A *well-ordered set* is a totally-ordered set such that every non-empty subset has a least element [Gr79, Page12]. The elements of a

sequence form a chain and conversely the elements of a chain can be placed in a sequence.

Definition C.1.3 (continuous). A monotonic operation g is *continuous* if and only if for every sequence in $\sqsubseteq$, $Q = \{q_i\}$, whenever $R = \{r_i\}$ is a sequence in $\sqsubseteq$ indexed by the same set as Q such that for all indexes i, $r_i \in g(\vec{q_i})$, then $\mathrm{lub}(R) \in g(\mathrm{lub}(Q))$.

The operations of an algebra are required to be continuous. The assumption that the carrier sets of all types except `IntStream` and `BoolStream` are flat domains is therefore restrictive, because there are some abstract types whose carrier sets cannot be considered flat domains if their operations are to be monotonic and continuous. For example, the carrier set of `IntStream` cannot be a flat domain, since then the `cons` operation would not be monotonic.

A set of values is *closed* if and only if for every chain $Q \subseteq D$, its least upper bound, $\mathrm{lub}(Q)$, is also in D.

It is also assumed that each set of possible results of an algebra's operations is closed with respect to the algebra's domain ordering $\sqsubseteq$. This assumption is necessary for the assignment of denotations to mutually recursive NOAL functions. This assumption also ensures that the set of possible results of each NOAL expression is closed and thus accords with the principle of finite observability [Bro86].

C.1.2 Semantics of Recursive Functions in NOAL

The semantics of systems of mutually recursive NOAL functions is given by several stages of approximation. First the semantics of systems that do not use angelic choice are defined. Approximations are obtained by eliminating erratic choice operators ($\square$) and textually expanding recursive calls. Erratic choices are turned into different expansions.

The denotation of a system of recursive function definitions is a tuple of mappings, each of which takes an algebra and returns a mapping from a tuple of arguments to a set of possible results.

The notation $\mathcal{F}[\![\vec{F}]\!]_i$ stands for the denotation of the i-th function in the system $\vec{F}$. If the i-th function is named f_i, then the abbreviation $\mathcal{F}[\![f_i]\!]$ means $\mathcal{F}[\![\vec{F}]\!]_i$.

The denotation of a system of function definitions does not depend on the environment, because in the body of a recursively defined NOAL function, there can be no free identifiers or function identifiers, besides those of the other recursively defined functions and the function's formal arguments.

Fix an algebra A. Let

$$\begin{aligned}&\texttt{fun } \mathtt{f}_1(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_1;\\ &\vdots\\ &\texttt{fun } \mathtt{f}_m(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_m\end{aligned}$$

be a mutually recursive system of NOAL function definitions, where the angelic choice operator ($\bigtriangledown$) does not occur in the E_j.

When eliminating erratic choice operators from an expression, one makes choices of what expressions to execute; each such choice is called a deterministic descendant. An expression E'' is a *deterministic descendant* of an expression E' if E'' does not contain the erratic choice operator ($\square$) and can be obtained from E' by replacing subexpressions of the form $\gamma_1 \square \gamma_2$ with either γ_1 or γ_2.

A family $D_{(j,i)}$ of expressions is called a *choice family* for the system of f_j if for each j, $D_{(j,0)}$ is a deterministic descendant of E_j, and $D_{(j,i+1)}$ is a deterministic descendant of $D_{(j,i)}$ with, for each k, the function abstract `(fun (`$\vec{\mathtt{x}_k} : \vec{\mathtt{S}_k}$`)` E_k`)` substituted for each occurrence of f_k in the body of $D_{(j,i)}$. The expression $D_{(j,i+1)}$ differs from $D_{(j,i)}$ in that one more recursion is unrolled, thus $D_{(j,i+1)}$ is a better approximation to one computation of f_j than $D_{(j,i)}$. For example, consider a system with one recursively defined function, where f_1 is the function *pick* defined by

```
fun pick (x:Int): Int =
    (x □ pick(add(x,1))).
```

There are infinitely many choice families for this example. One choice family for *pick* is for all i, $D_{(1,i)} = \mathtt{x}$. Another choice family has $D_{(1,i)} = D_{(1,1)}$ for all $i > 1$, where

$$\begin{aligned}D_{(1,0)} &= \textit{pick}\texttt{(add(x,1))}\\ D_{(1,1)} &= \texttt{(fun (x:Int) x) (add(x,1))}.\end{aligned}$$

There is also a choice family that has an occurrence of *pick* in every $D_{(1,i)}$.

As usual, an everywhere-$\bot$ function is the first approximation to recursive invocations in the $D_{(j,i)}$. For each j, let $\mathtt{G}_j$ be the function abstract of the form

`(fun (`$\vec{\mathtt{x}_j} : \vec{\mathtt{S}_j}$`) bottom[`$\mathtt{T}_j$`]).`

The least upper bounds of sequences of approximate results are used to define the meaning of a function for a given algebra. For each function index j, $DD_j(A)(\vec{q})$ denotes the set of all sequences of approximate results, $Q_j(A)(\vec{q})$, for all choice families for f_j. Given a choice family $D_{(j,i)}$, a sequence of approximate results is such that $Q_j(A)(\vec{q}) = \langle \hat{q}_i \rangle$, where for each i, $\hat{q}_i$ is a possible result of $\mathcal{M}[\![D_{(j,i)}[\vec{\mathtt{G}}/\vec{f}]]\!](A, \eta)$ and $\eta(\vec{\mathtt{x}_j}) = \vec{q}$. As Broy notes, there are such sequences in $\sqsubseteq$ because the deterministic language constructs (and each operation of A) are monotonic and because $D_{(j,i+1)}$ is derived from $D_{(j,i)}$ by unrolling another recursion. Note that $D_{(j,i)}[\vec{\mathtt{G}}/\vec{f}]$ is recursion-free.

For the *pick* example, $DD_1(A)(0)$ would be the set consisting of the sequence $\langle \bot, \bot, \bot, \ldots \rangle$ and all sequences in $\sqsubseteq$ of the form

$$\langle \underbrace{\bot, \bot, \ldots, \bot}_{n}, n, n, n, \ldots \rangle$$

for some $n \geq 0$.

The denotation $\mathcal{F}[\![f_j]\!]$ of a function definition that does not use angelic choice is defined as follows.

$$\begin{aligned}&\mathcal{F}[\![f_j]\!](A)(\vec{q})\\ &\quad \stackrel{\mathrm{def}}{=} \overline{\{\mathrm{lub}(Q_j(A)(\vec{q})) \mid Q_j(A)(\vec{q}) \in DD_j(A)(\vec{q})\}}\end{aligned} \tag{C.1}$$

That is, $\mathcal{F}[\![f_j]\!](A)(\vec{q})$ is the closure of the set of all the least upper bounds of all sequences in $\sqsubseteq$ from $DD_j(A)(\vec{q})$. The *closure* of a set Q, written $\overline{Q}$, is the smallest closed set that contains Q. Taking the closure ensures that the set of possible results is closed; it might otherwise be possible to form a sequence from the $\mathrm{lub}(Q_j(A)(\vec{q}))$ whose least upper bound was not in the set.

For the *pick* example, the possible results of *pick*`(0)` are determined as follows. The least upper bound of the sequence

$$\langle \bot, \bot, \bot, \ldots \rangle$$

is $\bot$. The least upper bound of a sequences of the form

$$\langle \bot, \bot, \bot, \ldots, n, n, n, \ldots \rangle$$

is n. Thus for all algebras A,

$$\begin{aligned} &\{\mathrm{lub}(Q_j(A)(0)) \mid Q_j(A)(0) \in DD_1(A)(0)\} \\ &\quad = \quad \{\bot, 0, 1, 2, 3, \ldots\}. \end{aligned}$$

This set is already closed in the $\sqsubseteq$ ordering (as the carrier set of `Int` is a flat domain) so it is the set of possible results.

The meaning of a system of recursive function definitions that uses angelic choice uses the meaning of a system that does not use angelic choice as a first approximation. Better approximations are obtained by using earlier approximations to evaluate recursive calls. The net effect is that each approximation uses angelic choice for deeper recursions than the previous approximation [Bro86, Page 19].

Let

$$\begin{aligned} &\texttt{fun } f_1(\vec{\texttt{x}_1} : \vec{\texttt{S}_1}) : \texttt{T}_1 = E_1; \\ &\vdots \\ &\texttt{fun } f_m(\vec{\texttt{x}_m} : \vec{\texttt{S}_m}) : \texttt{T}_m = E_m \end{aligned}$$

be a system of mutually recursive NOAL function definitions. Let $E_{(j,0)}$ be derived from E_j by replacing all occurrences of the angelic choice operator ($\bigtriangledown$) with the erratic choice operator ($\square$). For each j let $g_{(j,0)}$ refer to the definition of f_j with $E_{(j,0)}$ replacing E_j.

For example, consider the function

```
fun pick2 (x:Int): Int =
    (x ▽ pick2(add(x,1))).
```

For this example, the system with $\bigtriangledown$ replaced by $\square$ is

```
fun pick2 (x:Int): Int =
    (x □ pick2(add(x,1)))
```

and $g_{(1,0)}$ refers to this altered definition of *pick2*.

For each j, $\mathcal{F}[\![g_{(j,0)}]\!]$ gives meaning to recursive calls to f_j. The next approximation obtained in this way is called $\mathcal{F}[\![g_{(j,1)}]\!]$. In this way a family $\mathcal{F}[\![g_{(j,i)}]\!]$ for each j and i is defined as follows. For each natural number i,

$$\mathcal{F}[\![g_{(j,i+1)}]\!](A)(\vec{q}) \stackrel{\mathrm{def}}{=} \overline{\mathcal{M}[\![E_j]\!](A, \eta_i)}, \tag{C.2}$$

where

$$\eta_i(\vec{\texttt{x}_j}) = \vec{q} \tag{C.3}$$

and where for all k,

$$\eta_i(f_k) = \mathcal{F}[\![\vec{g}_{(k,i)}]\!](A). \tag{C.4}$$

So to find the possible results of $\mathcal{F}[\![g_{(j,i+1)}]\!](A)(\vec{q})$, one takes the possible results of E_j, which may use angelic choice, in an environment where the formals of the function definition f_j are bound to the arguments $\vec{q}$ and where $\mathcal{F}[\![\vec{g}_{(k,i)}]\!](A)$ is used as an approximation to f_k, for all k. (The only free identifiers in E_j are the $\vec{\texttt{x}}_j$, and the only free function identifiers are the f_k.) By construction, $\mathcal{F}[\![\vec{g}_{(j,i)}]\!](A)$ does i levels of recursion using angelic choice and then reverts to erratic choice.

For the *pick2* example,

$$\begin{aligned} &\mathcal{F}[\![g_{(1,0)}]\!](A)(0) \\ &\quad = \quad \{\bot, 0, 1, 2, 3, \ldots\} && \text{(C.5)} \\ &\mathcal{F}[\![g_{(1,1)}]\!](A)(0) \\ &\quad \stackrel{\mathrm{def}}{=} \quad \overline{\mathcal{M}[\![\texttt{x} \bigtriangledown \texttt{pick2(add(x,1))}]\!](A, \eta_0)} && \text{(C.6)} \\ &\quad = \quad \{0\} \cup (\{\bot, 1, 2, 3, \ldots\} \setminus \{\bot\}) && \text{(C.7)} \\ &\quad = \quad \{0, 1, 2, 3, \ldots\}. && \text{(C.8)} \end{aligned}$$

where $\eta_0(\texttt{x}) = 0$ and $\eta_0(pick2) = \mathcal{F}[\![g_{(1,0)}]\!](A)$. By the definition of angelic choice, $\bot$ is not a possible result of the expression `x ▽ pick2(add(x,1))` in η, because the only possible result of `x` is 0. The possible results of $\mathcal{F}[\![g_{(1,i)}]\!](A)(0)$ for all $i > 1$ are also $\{0, 1, 2, 3, \ldots\}$.

Following [Bro86, Page 19], for each j the meaning of f_j is

$$\mathcal{F}[\![f_j]\!](A)(\vec{q}) \stackrel{\mathrm{def}}{=} \bigcap_i \mathcal{F}[\![g_{(j,i)}]\!](A)(\vec{q}). \tag{C.9}$$

For the *pick2* example:

$$\begin{aligned} \mathcal{F}[\![pick2]\!](A)(0) &\stackrel{\mathrm{def}}{=} \bigcap_i \mathcal{F}[\![g_{(1,i)}]\!](A)(0) && \text{(C.10)} \\ &= \{0, 1, 2, 3, \ldots\}. && \text{(C.11)} \end{aligned}$$

C.2 The Substitution Property for Functions

The postponed proof of the substitution property for NOAL functions, Lemma 7.2.2, is given in this section.

Because the semantics of systems of mutually recursive function definitions involve closures and least upper bounds of sequences, it is convenient to first show that simulation relations are strongly monotonic and continuous.

Definition C.2.1 (strongly monotonic).
A binary relation $\ll$ between domains D_1 and D_2 is *strongly monotonic* if and only if for all $q_1, q_2 \in D_1$ and for all $r_1, r_2 \in D_2$, whenever $q_1 \sqsubset q_2$, $q_1 \ll r_1$, and $q_2 \ll r_2$, then $r_1 \sqsubset r_2$.

This definition is illustrated in Figure C.1. A family of relations $\mathcal{R}$ is strongly monotonic if each $\mathcal{R}_{\mathbf{T}}$ is a strongly monotonic relation.

The following lemma says that each simulation relation is strongly monotonic.

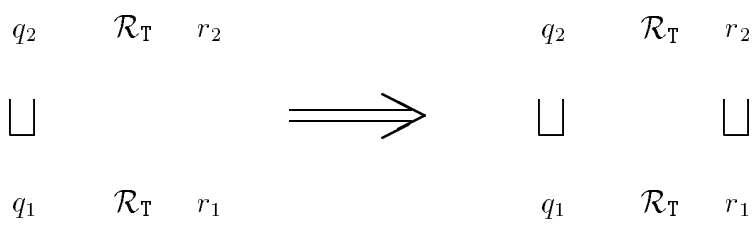

Figure C.1: Strong monotonicity of $\mathcal{R}_{\mathtt{T}}$.

Lemma C.2.2. Let Σ be a signature. Let C and A be Σ-algebras.

If $\mathcal{R}$ is a Σ-simulation relation between C and A, then $\mathcal{R}$ is strongly monotonic.

Proof: Let $\mathtt{T}$ be a sort. Suppose $q_1 \sqsubset q_2$, $q_1 \mathcal{R}_{\mathtt{T}} r_1$, and $q_2 \mathcal{R}_{\mathtt{T}} r_2$. Since $q_1 \sqsubset q_2$, either $q_1 = \bot$ and q_2 is proper or both q_1 and q_2 are proper elements of some visible type with a non-flat carrier set.

If $q_1 = \bot$ and q_2 is proper, then since $\mathcal{R}_{\mathtt{T}}$ is bistrict, $r_1 = \bot$ and r_2 is proper. So $r_1 \sqsubset r_2$.

If q_1 and q_2 are proper elements of a visible type then $q_1 = r_1$, and $q_2 = r_2$, because $\mathcal{R}$ is V-identical. ∎

The following lemma says that each simulation relation is continuous. A family of relations is continuous if it is continuous at each type.

Lemma C.2.3. Let Σ be a signature. Let A and B be Σ-algebras.

If $\mathcal{R}$ is a Σ-simulation relation between A and B, then $\mathcal{R}$ is continuous.

Proof: Let $\mathtt{T}$ be a sort. Let Q be a sequence in $\sqsubseteq$ of elements of A. Let R be a sequence in $\sqsubseteq$ of elements of B, indexed by the same set as Q, such that for all indexes i, $q_i \mathcal{R}_{\mathtt{T}} r_i$.

If the only elements of Q are $\bot$, then the only elements of R are $\bot$, since $\mathcal{R}_{\mathtt{T}}$ is bistrict; thus $\mathrm{lub}(Q) = \bot \mathcal{R}_{\mathtt{T}} \bot = \mathrm{lub}(R)$.

Otherwise, Q contains some proper elements. Since $\mathcal{R}_{\mathtt{T}}$ is bistrict, R also contains some proper elements and relates proper elements of Q to proper elements of R. Since the proper elements of Q are related by $\sqsubseteq$, the proper elements must all be either elements of a visible type or all elements of a non-visible type (by one of the assumptions about algebras).

If the proper elements of Q are elements of a visible type, $\mathtt{S}$, then since $\mathcal{R}$ is V-identical, for each i and each $q_i \in Q$, $q_i = r_i \in R$. Therefore $\mathrm{lub}(Q) = \mathrm{lub}(R)$. Since $\mathcal{R}$ is V-identical and $\mathtt{S}$ is a visible type, $\mathcal{R}_{\mathtt{T}}$ contains $\mathcal{R}_{\mathtt{S}}$ and hence the identity on the carrier set of $\mathtt{S}$. Therefore $\mathrm{lub}(Q)\, \mathcal{R}_{\mathtt{T}}\, \mathrm{lub}(R)$.

Likewise R must contain either visible or non-visible elements. If R contains proper visible elements, then since $\mathcal{R}$ is V-identical, for each i, $q_i = r_i$ and the result follows.

So the remaining case is that the proper elements of both Q and R are elements of a non-visible type. Hence they are all elements of a flat domain. So the least upper bound of Q must occur in Q. Similarly, the least upper bound of R must occur in R. Let j be an index such that $\mathrm{lub}(Q) = q_j \in Q$. Since R only contains elements of a flat domain, $\mathrm{lub}(R) = r_j$. ∎

The following lemma says that the closures of sets related by a strongly monotonic and continuous relation are related. This lemma is needed because closures are used in the semantics of systems of recursively defined NOAL functions.

Lemma C.2.4. Let D_1 and D_2 be domains. Let $\ll$ be a strongly monotonic and continuous relation between D_1 and D_2.

If $Q \subseteq D_1$ and $R \subseteq D_2$ are such that $Q \ll R$, then $\overline{Q} \ll \overline{R}$.

Proof: Suppose $\hat{q} \in \overline{Q}$, but $\hat{q} \notin Q$. Since $\hat{q} \in \overline{Q}$, there is some sequence in $\sqsubseteq$, Q_0, consisting of elements of Q such that $\mathrm{lub}(Q_0) = \hat{q}$. Let I be the well-ordered set that indexes Q_0 and let the elements of I be ordered by $\leq$. Since $Q \ll R$, a sequence in $\sqsubseteq$ from R such that $\mathrm{lub}(Q_0)$ is related by $\ll$ to its least upper bound can be defined inductively as follows. As the basis, let i_0 be the least element of the index set I. Since $Q \ll R$, there is some $r_{i_0} \in R$ such that $q_{i_0} \ll r_{i_0}$. For the inductive step, suppose that r_k is defined for all $k \in I$ such that $k \leq j$. Let i be the least element of I such that $j < i$; then r_i can be chosen as follows. If $q_j = q_i$, let $r_i = r_j$. Otherwise, if $q_j \sqsubset q_i$, let $r_i \in R$ be such that $q_i \ll r_i$. Such an r_i exists because $Q \ll R$. Since $\ll$ is strongly monotonic, if $q_j \sqsubset q_i$, then $r_j \sqsubset r_i$. Therefore $R_0 = \{r_i\}$ is a sequence in $\sqsubseteq$, such that for each i, $q_i \ll r_i$. Since $\ll$ is continuous, $\mathrm{lub}(Q_0) \ll \mathrm{lub}(R_0)$. Finally, by definition of closure, $\mathrm{lub}(R_0) \in \overline{R}$. ∎

The lemma below shows that the substitution property holds for recursively defined functions that do *not* use angelic choice. The case without angelic choice is treated first because this treatment parallels Broy's semantics for recursive function definitions.

Lemma C.2.5. Let

$$\begin{array}{l} \mathtt{fun}\ f_1(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_1; \\ \vdots \\ \mathtt{fun}\ f_m(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_m \end{array}$$

be a mutually recursive system of NOAL function definitions. Let Σ be a signature. Let A and B be Σ-algebras.

Suppose that $\bigtriangledown$ does not occur in the function bodies E_j. Suppose $\mathcal{R}$ is a Σ-simulation relation between

A and B. Then for each j from 1 to m,

$$\mathcal{F}[\![f_j]\!](A)\,\mathcal{R}_{\vec{\mathtt{S}_j}\to\mathtt{T}_j}\,\mathcal{F}[\![f_j]\!](B). \tag{C.12}$$

Proof: For each j, let $\mathtt{G}_j$ be the function abstract of the form

$$\texttt{(fun (}\vec{\mathtt{x}_j} : \vec{\mathtt{S}_j}\texttt{) bottom[}\mathtt{T}_j\texttt{])}.$$

Let $k \in \{1,\ldots,m\}$ be given. Let $\vec{q}$ and $\vec{r}$ have the same length as $\vec{\mathtt{x}_k}$ and be such that $\vec{q}\,\mathcal{R}_{\vec{\mathtt{S}_k}}\,\vec{r}$. Let $\eta_1(\vec{\mathtt{x}_k}) = \vec{q}$ and $\eta_2(\vec{\mathtt{x}_k}) = \vec{r}$. By construction, $\eta_1\,\mathcal{R}\,\eta_2$.

Let $D_{(j,i)}$ be a choice family, and let $Q_k(A)(\vec{q}) = \langle \hat{q}_i \rangle$ be a sequence in $\sqsubseteq$ such that for each i, $\hat{q}_i$ is a possible result of $\mathcal{M}[\![D_{(k,i)}[\vec{\mathtt{G}}/\vec{f}]]\!](A,\eta_1)$. Let $Q_k(B)(\vec{r}) = \langle \hat{r}_i \rangle$, be a sequence in $\sqsubseteq$, where for each i, $\hat{r}_i$ is a possible result of $\mathcal{M}[\![D_{(k,i)}[\vec{\mathtt{G}}/\vec{f}]]\!](B,\eta_2)$ and $\hat{q}_i\,\mathcal{R}_{\mathtt{T}_k}\,\hat{r}_i$. Such a sequence can be found, because $D_{(k,i)}[\vec{\mathtt{G}}/\vec{f}]$ is recursion-free, (thus Lemma 7.2.1 applies) and because $\mathcal{R}$ is strongly monotonic. Since $\mathcal{R}_{\mathtt{T}_k}$ is a continuous relation, $\mathrm{lub}(Q_k(A)(\vec{q}))\,\mathcal{R}_{\mathtt{T}_j}\,\mathrm{lub}(Q_k(B)(\vec{r}))$.

Let $DD_k(A)(\vec{q})$ denote the set of all such sequences $Q_k(A)(\vec{q})$ in $\sqsubseteq$ for all choice families and let $DD_k(B)(\vec{r})$ be similarly defined. By the above, for every $Q_k(A)(\vec{q}) \in DD_k(A)(\vec{q})$, there is some $Q_k(B)(\vec{r}) \in DD_k(B)(\vec{r})$ (obtained using the same choice family) such that

$$\mathrm{lub}(Q_k(A)(\vec{q}))\,\mathcal{R}_{\mathtt{T}_k}\,\mathrm{lub}(Q_k(B)(\vec{r})).$$

Therefore the following relationship holds.

$$\{\mathrm{lub}(Q_k(A)(\vec{q})) \mid Q_k(A)(\vec{q}) \in DD_k(A)(\vec{q})\}\;\mathcal{R}_{\mathtt{T}_k}\;\{\mathrm{lub}(Q_k(B)(\vec{r})) \mid Q_k(A)(\vec{r}) \in DD_k(B)(\vec{r})\} \tag{C.13}$$

Since these sets of least upper bounds are related by $\mathcal{R}_{\mathtt{T}_k}$ and $\mathcal{R}_{\mathtt{T}_k}$ is strongly monotonic and continuous, by Lemma C.2.4 the closures of these sets are related by $\mathcal{R}_{\mathtt{T}_k}$.

Therefore,

$$\mathcal{F}[\![f_k]\!](A)(\vec{q}) \tag{C.14}$$

$$\stackrel{\mathrm{def}}{=} \quad \overline{\{\mathrm{lub}(Q_k(A)(\vec{q})) \mid Q_k(A)(\vec{q}) \in DD_k(A)(\vec{q})\}}\;\mathcal{R}_{\mathtt{T}_k}\;\overline{\{\mathrm{lub}(Q_k(B)(\vec{r})) \mid Q_k(B)(\vec{r}) \in DD_k(B)(\vec{r})\}} \tag{C.15}$$

$$\stackrel{\mathrm{def}}{=} \quad \mathcal{F}[\![f_k]\!](B)(\vec{r}) \tag{C.16}$$

So for each j from 1 to m,

$$\mathcal{F}[\![f_j]\!](A)\,\mathcal{R}_{\vec{\mathtt{S}_j}\to\mathtt{T}_j}\,\mathcal{F}[\![f_j]\!](B).$$

∎

The following lemma begins the treatment of systems of function definitions that use angelic choice. Since the semantics of such systems is given by first replacing angelic choice with erratic choice, the following lemma describes how the set of possible results of an expression is affected by this substitution.

Lemma C.2.6. Let A be an algebra. Let X be a set of typed identifiers. Let $\eta : X \to |A|$ be an environment such that for each function identifier f, $\eta(f)$ is monotonic. Let γ be a NOAL expression.

Suppose γ' is derived from γ by replacing all the angelic choice ($\bigtriangledown$) operators in γ with erratic choice operators ($\square$). Then

$$\mathcal{M}[\![\gamma']\!](A,\eta) \quad \sqsubseteq_E \quad \mathcal{M}[\![\gamma]\!](A,\eta). \tag{C.17}$$

Proof: (by induction on the structure of NOAL expressions.)

As a basis, if γ is an identifier or `bottom[T]` for some type `T` then the result is trivial.

For the inductive step, suppose that the result holds for each subexpression. As Broy points out [Bro86, Theorem 3.2], the meaning of each expression except angelic choice that has subexpressions $\gamma_1,\ldots,\gamma_n$ has the form

$$\mathcal{M}[\![\mathrm{expr}(\vec{\gamma})]\!](A,\eta) = \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{\gamma}]\!](A,\eta)} h(\vec{q})$$

for some monotonic set-valued function h. In particular, each operation of an algebra is monotonic and by hypothesis, for each function identifier f, $\eta(f)$ is monotonic in $\sqsubseteq_E$. If $\mathrm{expr}(\vec{\gamma'})$ is derived from $\mathrm{expr}(\vec{\gamma})$ by replacing all occurrences of $\bigtriangledown$ with $\square$, then by the inductive hypothesis we have $\mathcal{M}[\![\vec{\gamma'}]\!](A,\eta) \sqsubseteq_E \mathcal{M}[\![\vec{\gamma}]\!](A,\eta)$. Therefore,

$$\mathcal{M}[\![\mathrm{expr}(\vec{\gamma'})]\!](A,\eta) = \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{\gamma'}]\!](A,\eta)} h(\vec{q}) \tag{C.18}$$

$$\sqsubseteq_E \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{\gamma}]\!](A,\eta)} h(\vec{q}) \tag{C.19}$$

$$= \mathcal{M}[\![\mathrm{expr}(\vec{\gamma})]\!](A,\eta). \tag{C.20}$$

Finally, consider the expressions $\gamma_1 \bigtriangledown \gamma_2$ and the derived expression $\gamma_1' \square \gamma_2'$. By definition of NOAL,

$$\mathcal{M}[\![\gamma_1' \square \gamma_2']\!](A,\eta)$$

$$\stackrel{\mathrm{def}}{=} \quad \mathcal{M}[\![\gamma_1']\!](A,\eta) \cup \mathcal{M}[\![\gamma_2']\!](A,\eta) \tag{C.21}$$

$$\sqsubseteq_E \quad \mathcal{M}[\![\gamma_1]\!](A,\eta) \cup \mathcal{M}[\![\gamma_2]\!](A,\eta) \tag{C.22}$$

$$\stackrel{\mathrm{def}}{=} \quad \mathcal{M}[\![\gamma_1 \square \gamma_2]\!](A,\eta) \tag{C.23}$$

$$\sqsubseteq_E \quad \mathcal{M}[\![\gamma_1 \bigtriangledown \gamma_2]\!](A,\eta), \tag{C.24}$$

which holds because $\mathcal{M}[\![\gamma_1 \square \gamma_2]\!](A,\eta)$ differs from $\mathcal{M}[\![\gamma_1 \bigtriangledown \gamma_2]\!](A,\eta)$ in that the former may contain $\perp$ when the latter does not. ∎

The next lemma shows that the substitution property holds for systems of recursively defined functions that may use angelic choice. This lemma is the same as Lemma 7.2.2.

Lemma C.2.7. Let

$$\texttt{fun } \mathtt{f}_1(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_1;$$
$$\vdots$$
$$\texttt{fun } \mathtt{f}_m(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_m$$

be a mutually recursive system of NOAL function definitions.

Suppose $\mathcal{R}$ is a Σ-simulation relation between Σ-algebras A and B. Then for each j from 1 to m,

$$\mathcal{F}[\![f_j]\!](A)\,\mathcal{R}_{\vec{\mathbf{S}}_j \to \mathbf{T}_j}\,\mathcal{F}[\![f_j]\!](B). \qquad \text{(C.25)}$$

Proof: Let $E_{(j,0)}$ be derived from E_j by replacing all occurrences of the angelic choice operator ($\bigtriangledown$) with the erratic choice operator ($\square$). For each j let $g_{(j,0)}$ refer to the definition of f_j with $E_{(j,0)}$ replacing E_j. By Lemma C.2.5, for each j from 1 to m,

$$\mathcal{F}[\![g_{(j,0)}]\!](A)\,\mathcal{R}_{\vec{\mathbf{S}}_j \to \mathbf{T}_j}\,\mathcal{F}[\![g_{(j,0)}]\!](B). \qquad \text{(C.26)}$$

The discussion of the semantics of recursive systems above inductively defines a family of approximations for the meaning of $\vec{f}$ in A, $\mathcal{F}[\![g_{(j,i)}]\!](A)$, and a corresponding family for the meaning of $\vec{f}$ in B, $\mathcal{F}[\![g_{(j,i)}]\!](B)$. The proof proceeds by showing two properties of these families of approximations.

The first property is that for all j from 1 to m,

$$\mathcal{F}[\![g_{(j,i)}]\!](A)\,\mathcal{R}_{\vec{\mathbf{S}}_j \to \mathbf{T}_j}\,\mathcal{F}[\![g_{(j,i)}]\!](B). \qquad \text{(C.27)}$$

This follows by induction on i, using Lemma 7.2.1 and Lemma C.2.4.

The second property is that for all natural numbers i, for all j from 1 to m, and for all arguments $\vec{q}$ from the algebra B,

$$\mathcal{F}[\![g_{(j,i)}]\!](A)(\vec{q}) \quad \sqsubseteq_E \quad \mathcal{F}[\![g_{(j,i+1)}]\!](B)(\vec{q}). \qquad \text{(C.28)}$$

Intuitively, this should hold because $\mathcal{F}[\![g_{(j,i+1)}]\!](B)$ uses angelic choice for one more recursion than does $\mathcal{F}[\![g_{(j,i)}]\!](B)$.

The second property is proved by induction on i. For the basis, let j be fixed and let $\vec{q}$ be given. Let η_0 be an environment such that $\eta_0(\vec{\mathbf{x}_j}) = \vec{q}$ and for all $k \in \{1, \ldots, m\}$, $\eta_0(f_k) = \mathcal{F}[\![g_{(k,0)}]\!](B)$. Each $\eta_0(f_k)$ is monotonic, because the bodies of the $g_{(k,0)}$ do not use angelic choice. By construction of the $\mathcal{F}[\![g_{(k,0)}]\!](B)$,

$$\mathcal{F}[\![g_{(j,0)}]\!](B)(\vec{q}) = \mathcal{M}[\![E_{(j,0)}]\!](B, \eta_0). \qquad \text{(C.29)}$$

By Lemma C.2.6,

$$\mathcal{M}[\![E_{(j,0)}]\!](B, \eta_0) \quad \sqsubseteq_E \quad \mathcal{M}[\![E_j]\!](B, \eta_0) \quad \text{(C.30)}$$

since $E_{(j,0)}$ is derived from E_j by replacing all the angelic choice operators with erratic choice operators. Since by construction, the set $\mathcal{M}[\![E_{(j,0)}]\!](B, \eta_0)$ is closed,

$$\begin{aligned} \overline{\mathcal{M}[\![E_{(j,0)}]\!](B, \eta)} &= \mathcal{M}[\![E_{(j,0)}]\!](B, \eta) && \text{(C.31)} \\ &\sqsubseteq_E \mathcal{M}[\![E_j]\!](B, \eta) && \text{(C.32)} \\ &\sqsubseteq_E \overline{\mathcal{M}[\![E_j]\!](B, \eta)}. && \text{(C.33)} \end{aligned}$$

By definition,

$$\mathcal{F}[\![g_{(j,1)}]\!](B)(\vec{q}) = \overline{\mathcal{M}[\![E_j]\!](B, \eta_0)}. \qquad \text{(C.34)}$$

So combining the above,

$$\mathcal{F}[\![g_{(j,0)}]\!](B)(\vec{q}) \quad \sqsubseteq_E \quad \mathcal{F}[\![g_{(j,1)}]\!](B)(\vec{q}). \qquad \text{(C.35)}$$

For the inductive step, assume that for all j and all $\vec{q}$,

$$\mathcal{F}[\![g_{(j,i-1)}]\!](B)(\vec{q}) \quad \sqsubseteq_E \quad \mathcal{F}[\![g_{(j,i)}]\!](B)(\vec{q}). \qquad \text{(C.36)}$$

Let η_{i-1} be an environment such that for all $k \in \{1, \ldots, m\}$, $\eta_{i-1}(f_k) = \mathcal{F}[\![g_{(k,i-1)}]\!](B)$ and such that $\eta_{i-1}(\vec{\mathbf{x}_j}) = \vec{q}$. Let η_i be an environment such that for all $k \in \{1, \ldots, m\}$, $\eta_i(f_k) = \mathcal{F}[\![g_{(k,i)}]\!](B)$ and such that $\eta_i(\vec{\mathbf{x}_j}) = \vec{q}$. By definition,

$$\begin{aligned} \mathcal{F}[\![g_{(j,i+1)}]\!](B)(\vec{q}) &= \overline{\mathcal{M}[\![E_j]\!](B, \eta_i)} && \text{(C.37)} \\ \mathcal{F}[\![g_{(j,i)}]\!](B)(\vec{q}) &= \overline{\mathcal{M}[\![E_j]\!](B, \eta_{i-1})}. && \text{(C.38)} \end{aligned}$$

Furthermore, by induction on the structure of NOAL expressions (as in Lemma C.2.6), the induction hypothesis can be used to show that

$$\mathcal{M}[\![E_j]\!](B, \eta_{i-1}) \quad \sqsubseteq_E \quad \mathcal{M}[\![E_j]\!](B, \eta_i) \quad \text{(C.39)}$$

So the second property (Formula C.28) holds.

Returning to the proof o fthe main result, let j be fixed and suppose $\vec{q_j}\ \mathcal{R}_{\vec{\mathbf{S}}_j}\ \vec{r}_j$. By definition of NOAL the following hold.

$$\mathcal{F}[\![f_j]\!](A)(\vec{q_j}) \stackrel{\text{def}}{=} \bigcap_i \mathcal{F}[\![g_{(j,i)}]\!](A)(\vec{q_j}) \quad \text{(C.40)}$$

$$\mathcal{F}[\![f_j]\!](B)(\vec{r_j}) \stackrel{\text{def}}{=} \bigcap_i \mathcal{F}[\![g_{(j,i)}]\!](B)(\vec{r_j}) \quad \text{(C.41)}$$

Suppose $q \in \mathcal{F}[\![f_j]\!](A)(\vec{q_j})$. Then for all i, $q \in \mathcal{F}[\![g_{(j,i)}]\!](A)(\vec{q_j})$. Since the first property (Formula C.27) holds for each i, there is some $r_i \in \mathcal{F}[\![g_{(j,i)}]\!](B)(\vec{r_j})$ such that $q\ \mathcal{R}_{\mathbf{T}_j}\ r_i$. There are several cases.

- If $q = \bot$, then since $\mathcal{R}_{\mathbf{T}_j}$ is bistrict, q can only be related to $\bot$. So each r_i is $\bot$, and thus $\bot \in \mathcal{F}[\![f_j]\!](B)(\vec{r_j})$. So q is related to some element of $\mathcal{F}[\![f_j]\!](B)(\vec{r_j})$ by $\mathcal{R}_{\mathbf{T}_j}$.

- If q is a proper instance of a visible type, then since $\mathcal{R}$ is V-identical, each $r_i = q$, and thus $q \in \mathcal{F}[\![f_j]\!](B)(\vec{r_j})$.

- If q is a proper instance of some non-visible type, then each of the r_i must be instances of a non-visible type as well, since $\mathcal{R}$ is V-identical. Suppose $r_0 \notin \mathcal{F}[\![g_{(j,1)}]\!](B)(\vec{r_j})$, then $r_0 \neq r_1$ and hence $r_0 \sqsubset r_1$, since $\mathcal{F}[\![g_{(j,0)}]\!](B)(\vec{r_j}) \sqsubseteq_E \mathcal{F}[\![g_{(j,1)}]\!](B)(\vec{r_j})$. But since r_0 and r_1 are elements of a non-visible type, they are elements of a flat domain, and therefore $r_0 = \bot$. But this contradicts the assumption that q is proper, since $\mathcal{R}$ is bistrict. So it must be that $r_0 = r_1$. By induction on i, it follows that for all natural numbers i, $r_i = r_0$. Therefore $r_0 \in \mathcal{F}[\![f_j]\!](B)(\vec{r_j})$ and $q \mathcal{R}_{\mathbf{T}_j} r_0$.

So whenever $\vec{q_j}\ \mathcal{R}_{\vec{\mathtt{S}_j}}\ \vec{r_j}$,

$$\mathcal{F}[\![f_j]\!](A)(\vec{q_j})\ \mathcal{R}_{\mathtt{T}_j}\ \mathcal{F}[\![f_j]\!](B)(\vec{r_j}). \tag{C.42}$$

Therefore for all j,

$$\mathcal{F}[\![f_j]\!](A)\ \mathcal{R}_{\vec{\mathtt{S}_j}\to\mathtt{T}_j}\ \mathcal{F}[\![f_j]\!](B). \tag{C.43}$$

∎

References

[Ada83] American National Standards Institute. *Reference Manual for the Ada Programming Language*, February 1983. ANSI/MIL-STD 1815A. Also published by Springer-Verlag as LNCS 155.

[AK84] Hassan Ait-Kaci. *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures*. PhD thesis, University of Pennsylvania, 1984.

[Ame89] Pierre America. A Behavioural Approach to Subtyping in Object-Oriented Programming Languages. Technical Report 443, Philips Research Laboratories, Nederlandse Philips Bedrijven B. V., January 1989.

[BDMN73] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *SIMULA Begin*. Auerbach Publishers, Philadelphia, Penn., 1973.

[BHJ+87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. *ACM SIGPLAN Notices*, 21(11):78–86, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[BL88] Kim B. Bruce and Giuseppe Longo. A Modest Model of Records, Inheritance, and Bounded Quantification. In Y. Gurevich, editor, *Logic in Computer Science*, pages 38–50. IEEE, July 1988.

[Bro86] Manfred Broy. A Theory for Nondeterminism, Parallelism, Communication, and Concurrency. *Theoretical Computer Science*, 45(1):1–61, 1986.

[BW86] Kim B. Bruce and Peter Wegner. An Algebraic Model of Subtypes in Object-Oriented Languages (Draft). *ACM SIGPLAN Notices*, 21(10), October 1986.

[BW87a] Kim B. Bruce and Peter Wegner. Algebraic and Lambda Calculus Models of Subtype and Inheritance (Extended Abstract). Working paper?, 1987.

[BW87b] Kim B. Bruce and Peter Wegner. An Algebraic Model of Subtype and Inheritance. To appear in *Database Programming Languages*, Francois Bancilhon and Peter Buneman (editors), Addison-Wesley, Reading, Mass., August 1987.

[Car84] Luca Cardelli. A Semantics of Multiple Inheritance. In D. B. MacQueen G. Kahn and G. Plotkin, editors, *Semantics of Data Types: International Symposium, Sophia-Antipolis, France*, volume 173 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, New York, N.Y., June 1984. A revised version of this paper appears in *Information and Computation*, volume 76, numbers 2/3, pages 138–164, February/March 1988.

[Car88] Luca Cardelli. Structural Subtyping and the Notion of Power Type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 70–79. ACM, January 1988.

[Car89] Luca Cardelli. Typeful Programming. Research Report 45, Digital Equipement Corporation, Systems Research Center, May 1989.

[CCH+89] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-Bounded Polymorphism for Object-Oriented Programming. In *Fourth International Conference on Functional Programming and Computer Architecture*. ACM, September 1989. Also technical report STL-89-5, from Software Technology Laboratory, Hewlett-Packard Laboratories.

[Che89] Jolly Chen. The Larch/Generic Interface Language. Technical report, Massachusetts Institute of Technology, May 1989. The author's Bachelor's thesis.

[CM89] Luca Cardelli and John C. Mitchell. Operations on Records. In *Fifth International Conference on Mathematical Foundations of Programming Semantics*, March 1989.

[Cox86] Brad J. Cox. *Object Oriented Programming: an Evolutionary Approach*. Addison-Wesley Publishing Co., Reading, Mass., 1986.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, New York, N.Y., 1985.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., Orlando, Florida, 1972.

[FGJM85] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and Jose Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, January 1985.

[GH78] J. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10(1):27–52, 1978.

[GH86a] J. V. Guttag and J. J. Horning. A Larch Shared Language Handbook. *Science of Computer Programming*, 6:135–157, 1986.

[GH86b] J. V. Guttag and J. J. Horning. Report on the Larch Shared Language. *Science of Computer Programming*, 6:103–134, 1986.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in Five Easy Pieces. Technical Report 5, Digital Systems Research Center, July 1985.

[GM87] Joseph A. Goguen and Jose Meseguer. Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems. Technical Report CSLI-87-92, Center for the Study of Language and Information, March 1987.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, N.Y., 1979. The second author is listed on the cover as Arthur J. Milner, which is clearly a mistake.

[Gog84] Joseph A. Goguen. Parameterized Programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Co., Reading, Mass., 1984.

[Goo75] J. B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[Gr79] George Grätzer. *Universal Algebra*. Springer-Verlag, New York, N.Y., second edition, 1979.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation*. Addison-Wesley Publishing Co., Reading, Mass., 1983.

[Gut80] John Guttag. Notes on Type Abstractions (Version 2). *IEEE Transactions on Software Engineering*, SE-6(1):13–23, January 1980. Version 1 in *Proceedings Specifications of Reliable Software*, Cambridge, Mass., IEEE, April, 1979.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hud89] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[JL76] Anita K. Jones and Barbara H. Liskov. A Language Extension for Controlling Access to Shared Data. *IEEE Transactions on Software Engineering*, SE-2(4):277–285, December 1976.

[JL78] Anita K. Jones and Barbara H. Liskov. A Language Extension for Expressing Constraints on Data Access. *Communications of the ACM*, 21(5):358–367, May 1978.

[JM88] Lalita A. Jategaonkar and John C. Mitchell. ML with Extended Patterm Matching and Subtypes (preliminary version). In *ACM Conference on LISP and Functional Programming, Snowbird, Utah*, pages 198–211, July 1988.

[Jon86] Cliff B. Jones. Program Specification and Verification in VDM. Technical Report

UMCS-86-10-5, Department of Computer Science, University of Manchester, November 1986.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison Wesley, Reading, Mass., 1989.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, N.Y., 1981.

[LaL89] Wilf R. LaLonde. Designing Families of Data Types Using Exemplars. *ACM Transactions on Programming Languages and Systems*, 11(2):212–248, April 1989.

[LDH+87] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Gary Leavens, Robert Scheifler, and William Weihl. Argus Reference Manual. Technical Report 400, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1987. An earlier version appeared as Programming Methodology Group Memo 54 in March 1987.

[Lea88] Gary T. Leavens. *Verifying Object-Oriented Programs that use Subtypes.* PhD thesis, Massachusetts Institute of Technology, December 1988. Published as MIT/LCS/TR-439 in February 1989.

[Lea89] Gary T. Leavens. Verifying Object-Oriented Programs that use Subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1989. The author's Ph.D. thesis.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *ACM SIGPLAN Notices*, 21(11):214–223, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[Lis88] Barbara Liskov. Data Abstraction and Hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[LL85] Gary T. Leavens and Barbara Liskov. The Name Clash Problem and a Proposed Solution. DSG Note 130, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1985.

[LS79] Barbara H. Liskov and Alan Snyder. Exception Handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979.

[LTP86] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An Exemplar Based Smalltalk. *ACM SIGPLAN Notices*, 21(11):322–330, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[LW90] Gary T. Leavens and William E. Weihl. Reasoning about Object-oriented Programs that use Subtypes (extended abstract). Technical Report 90-03, Iowa State University, Department of Computer Science, March 1990. To appear in ECOOP/OOPSLA '90.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., 1988.

[Mit86] John C. Mitchell. Representation Independence and Data Abstraction (preliminary version). In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 263–276. ACM, January 1986.

[MS82] D. B. MacQueen and Ravi Sethi. A Semantic Model of Types for Applicative Languages. In *ACM Symp. on LISP and Functional Programming*, pages 243–252. ACM, 1982.

[Nip86] Tobias Nipkow. Non-deterministic Data Types: Models and Implementations. *Acta Informatica*, 22(16):629–661, March 1986.

[Nip87] Tobias Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data Types.* PhD thesis, University of Manchester, May 1987.

[Par72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

[Rey80] John C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, January 1980.

[Rey85] John C. Reynolds. Three Approaches to Type Structure. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin. Volume 1: Colloquium on Trees in Algebra and Programming (CAAP '85)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, New York, N.Y., March 1985.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. *ACM SIGPLAN Notices*, 21(11):9–16, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, Mass., 1986.

[SCW85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis Object-Based Environment: Language Reference Manual. Technical Report DEC-TR-372, Eastern Research Lab, Digital Equipment Corp., Hudson, Mass., November 1985.

[SLU89] Lynn Andrea Stein, Henry Lieberman, and David Ungar. A Shared View of Sharing: The Treaty of Orlando. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 3, pages 31–48. Addison-Wesley Publishing Co., Reading, Mass., 1989.

[Sny86a] Alan Snyder. CommonObjects: An Overview. Technical Report STL-86-13, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, California, June 1986.

[Sny86b] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *ACM SIGPLAN Notices*, 21(11):38–45, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[ST85] Donald Sannella and Andrzej Tarlecki. On Observational Equivalence and Algebraic Specification. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin. Volume 1: Colloquium on Trees in Algebra and Programming (CAAP '85)*, volume 185 of *Lecture Notes in Computer Science*, pages 308–322. Springer-Verlag, New York, N.Y., March 1985.

[Sta85] R. Statman. Logical Relations and the Typed λ-Calculus. *Information and Control*, 65(2/3):85–97, May/June 1985.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Co., Reading, Mass., 1986. Corrected reprinting, 1987.

[Sym84] Symbolics, Inc. *Lisp Machine Manual.* Cambridge, Mass., March 1984. Eight volumes.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc Polymorphism less ad hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[Win83] Jeannette Marie Wing. A Two-Tiered Approach to Specifying Programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Win87] Jeannette M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR 90-09
Submission Date: July 5, 1990